AD-A047 393 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS F/G 9/2
COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE IMPLEMENTATION--ETC(U)
MAY 77 C R MORGAN, A EVANS DCA100-76-C-0051
UNCLASSIFIED BBN-3533 SBIE-AD-E100006 NL

1 OF 2
AD
A047393

R-3533

DEFENSE COMMUNICATIONS ENGINEERING CENTER

REPORT NO. 3533

COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE IMPLEMENTATION AND USAGE

MAY 1977

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREPARED FOR THE DEFENSE COMMUNICATIONS AGENCY
BY BOLT BERANEK & NEWMAN INC.

Report No. 3533 Bolt Beranek and Newman Inc.

COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE IMPLEMENTATION AND USAGE

C. Robert Morgan
Arthur Evans, Jr.

2 May 1977

Submitted to:

Mr. Paul M. Cohen, R-810
Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, Virginia 22090

The research reported herein was supported by the Defense Communications Agency of the Department of Defense, contract No. DCA 100-76-C-0051.

Reston, Virginia
May 4, 1977

The Defense Communications Engineering Center is currently engaged in the development of a capability to efficiently and economically produce software for the Defense Communications System. The objectives of this effort are to provide management control for this software, to ensure that this software meets the requirements for the system factors such as speed, reliability and security, and to insure that such software is developed, tested, verified and maintained in a manner that is consistent with the evolving state-of-the-art.

This is a report covering work which has been performed by BBN in support of such efforts.

Comments on this report from all government, industry, and university sources are encouraged, and will be greatly appreciated. We look at such comments as a cource of information for our future studies in this area. Please send all comments to me at the address below or to my ARPANET address.

Paul M. Cohen, Code R810
Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, Virginia 22090

ARPANET Address: COHEN at BBN-TENEXB

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(14) 1. REPORT NUMBER: BBN [redacted]-3533

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE IMPLEMENTATION AND USAGE.

(13) 5. TYPE OF REPORT & PERIOD COVERED: Final Report: May [redacted]76-May [redacted]77

6. PERFORMING ORG. REPORT NUMBER: BBN Report No. 3533

(10) 7. AUTHOR(s): C. Robert Morgan, Arthur Evans, Jr.

(15) 8. CONTRACT OR GRANT NUMBER(s): DCA100-76-C-0051

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: n/a

(11) 11. CONTROLLING OFFICE NAME AND ADDRESS:
Defense Communications Engineering Ctr.
1860 Wiehle Avenue Code R800
Reston, Virginia 22090

(12) 12. REPORT DATE: 2 May [redacted]77

13. NUMBER OF PAGES: iii +174 (180 p.)

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): n/a

15. SECURITY CLASS. (of this report): unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

(18) SBIE (19) AD-E100 116

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

n/a

060 100

18. SUPPLEMENTARY NOTES

This report is an extension of BBN Report No. 3261, "Development of a Communications Oriented Language, Part 1 of Final Report", 20 March 1976. Appendices A, B, and C of BBN Report No. 3261, and all of BBN Report No. 3261, Part II are now obsolete.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Communications Oriented Language, COL, High Order Language, HOL, Language Evaluation, Language Design, Compiler Design, Communications Systems, Computer Communications, DOD1, IRONMAN.

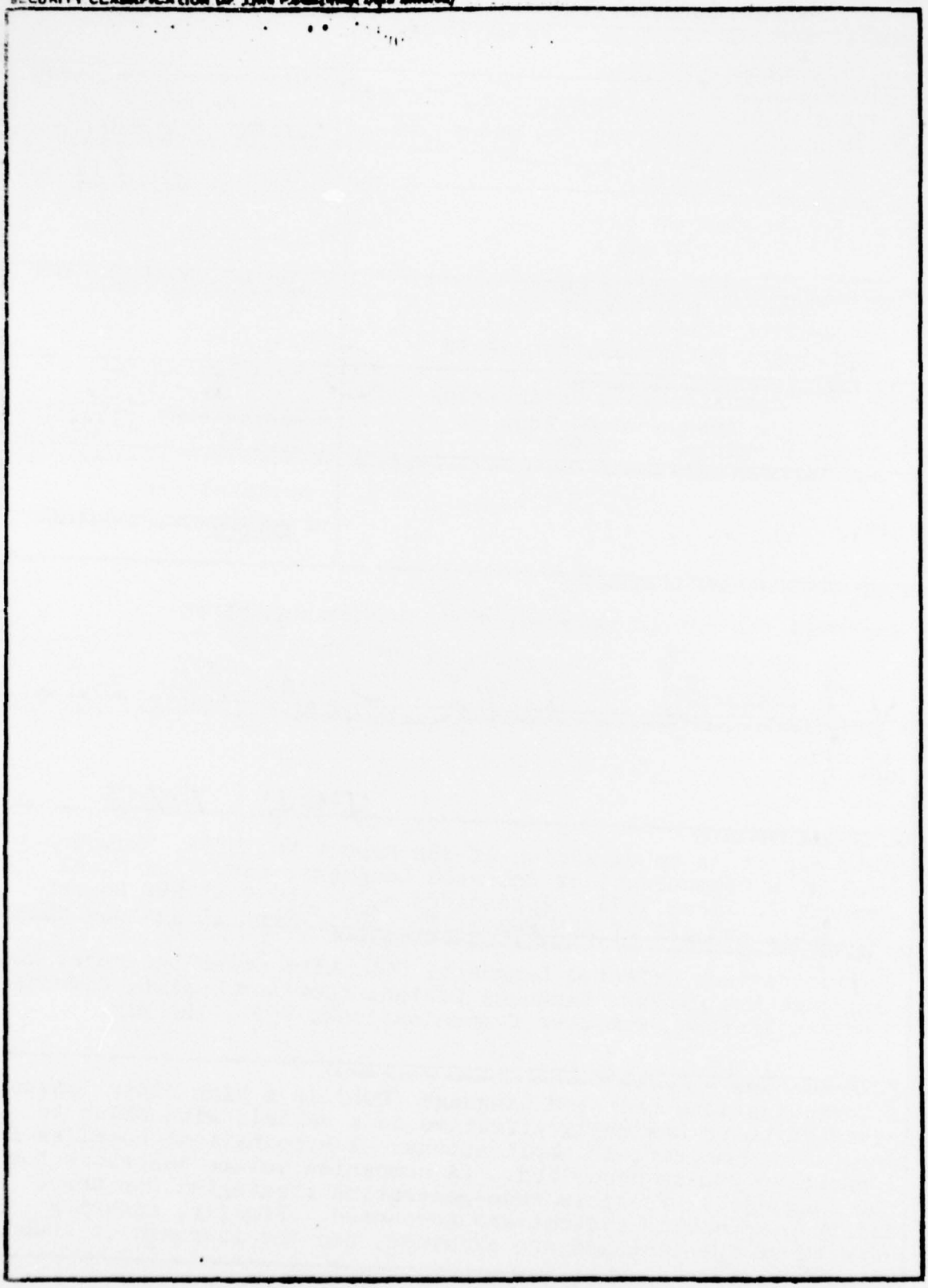
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A Communications Oriented Language (COL) is a high order language designed to be maximally effective as a vehicle with which to program communications applications. A hypothetical compiler for a proposed COL is described. (A companion volume describes the COL language.) Possible code-generation strategies for three different computer systems are presented. Finally, security aspects of the language are examined, and the language is compared with CS-4.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



Abstract

A Communications Oriented Language (COL) is a high order language designed to be maximally effective as a vehicle with which to program communications applications. A hypothetical compiler for a proposed COL is described. (A companion volume describes the COL language.) Possible code-generation strategies for three different computer systems are presented. Finally, security aspects of the language are examined, and the language is compared with CS-4.

Acknowledgments

The research reported here was performed by Arthur Evans, Jr., and C. Robert Morgan. Professor Edgar T. Irons of the Computer Science Department of Yale University made important contributions to the early part of the compiler design. Michael Condry of Yale wrote the first draft of Chapter 14 of this document, and Spencer Rugaber of Yale wrote the first draft of Chapter 16.

We are particularly grateful to Paul M. Cohen of the Defense Communications Engineering Center of DCA, both for making technical contributions to the design of the language and for providing us an unhindered opportunity to perform this work.

Table of Contents

Introduction

The United States Department of Defense (DoD) spends about $3 billion per year on computer software, exclusive of those computer functions commonly referred to as automatic data processing. There has been a growing realization that this cost could be decreased significantly, indeed dramatically, by suitable use of a high order language (HOL). The Defense Communications Agency (DCA) has long been involved in programming digital computers to perform communications applications, invariably in assembly language with its high attendant costs in both programmer productivity and difficulty of maintenance. In an attempt to reduce these costs, DCA has requested that Bolt Beranek and Newman Inc. (BBN) develop a high order language oriented towards communications applications. In an earlier contract with DCA (Contract No. 100-DCA-75-C-0051), BBN has studied the needs of communications programming and investigated existing languages to determine the extent to which they meet these needs. The results of this study are reported in BBN Report No. 3261, "Development of a Communications Oriented Language", 20 March 1976. The conclusion reported there was that no single language met adequately the needs of communications applications. BBN therefore proposed a communications oriented language (COL) specifically designed to meet the needs of such communications programming.

In the present contract, for which this document is part of the final report, BBN has continued to refine the design of the COL. The input for this process has come from several sources:

- We have designed a hypothetical compiler for the COL. The compiler has been designed with careful attention to making as much of it as possible totally independent of the object machine. The design of the compiler provided important feedback to the design of the language. In particular, an immensely important criterion in language design throughout the two contracts has been that object code must be highly efficient. By designing the compiler, we were able to isolate those language features which were incompatible with this need for efficiency and to remove such features from the language.

- BBN programmers not affiliated with the COL contract were asked to study the COL documentation and to program three modules in the COL, modules which they had previously coded in assembly language. They reported to us their impressions of the language and their suggestions for improvement, both of the language and of the documentation. (Both sets of suggestions were extremely valuable to us.) Our study of the code which they wrote

helped us to understand better the interaction between COL features and communications applications.

- Three programmers not previously familiar with the COL were asked to study the documentation. They did so and provided us with further feedback about both the language and the document.

- We compared the COL with the language CS-4, studying those features which were in either language and not in the other. This process gave us further suggestions for changes in the COL.

- We thought through to a limited extent the implications of the need for security in the object program on the design of the language. We were able to commit only limited resources to this part of the study and were not able to undertake it until fairly late in the contract. Nonetheless, this study pointed out to us some features of the COL which are not fully compatible with security. Were there more time to pursue this matter, there are certain aspects of the COL which we might change as a result of this study.

. Throughout the two contracts we have kept cognizant of the present efforts of the High Order Language Working Group (HOLWG) within DoD to develop high order languages for DoD usage. In the final report of the previous contract we compared the COL with the so-called "TINMAN" report. This final report contains a comparison of the COL with the "IRONMAN" report. In general, our differences with the "IRONMAN" are either very minor or are based on our feeling that the specialized needs of communications differ in some ways from the needs perceived by HOLWG. We have therefore designed a language with slightly different characteristics.

All of these matters are reported in the present final report.

This final report is presented in two volumes, of which this is the first. This volume contains a description of the hypothetical COL compiler, as well as certain other items described below. Volume II is a description of the COL language and is referred to hereafter as "the COL Manual". The reader is advised to read the COL Manual before attempting this volume, since this volume has been written with the assumption that the reader is familiar with the COL language.

This volume is organized as follows: The first 12 chapters contain the description of the hypothetical COL compiler; organization of these chapters is presented in chapter 1. Chapter 13 describes how the COL might be implemented on the DECsystem-20 or the Pluribus system, and chapter 14 describes a possible implementation on the IBM 360 (or 370). Chapter 15 describes the impact of security on the COL, and chapter 16 contrasts the COL with the language CS-4.

Throughout the entire effort close liaison was maintained with the DCA contracting officer's representative (COR). The direction in which the work proceeded and the relative emphasis given to the various tasks were the result of decisions reached mutually between DCA and the study team. However, although the status of our work was periodically reviewed by DCA, the conclusions and recommendations presented in this report should be considered solely those of BBN. Preliminary oral presentations to DCA were given approximately half way through the task and shortly before its conclusion. Feedback from the first presentation has influenced this report. Further, it is a pleasure to acknowledge the assistance and suggestions made to the study team by the COR.

Chapter 1

Introduction to the Compiler Design

The COL language has been designed by BBN under contract from the Defense Communications Agency to provide the features and advantages of high level languages to communications programming. In communications programming, one of the prime requirements is that efficient code be generated by whatever compiler or assembler is used. Thus part of the COL design effort is the design of a compiler for the COL.

The compiler design has been performed to determine three things: first, to guarantee that an efficient compiler is possible; second, to find those parts of the language which are difficult to implement and to modify them in order to improve the efficiency of the language; and third, to determine how a compiler for the COL could be written which would be easily portable from one machine to another.

The compiler will be divided into functional units, called phases, as follows:

- command decoding
- lexical analysis
- syntax analysis
- semantic analysis
- machine independent local optimization
- machine independent global optimization
- register allocation

- code generation
- peephole optimization
- assembly of object code

These phases are grouped into units called passes, each pass corresponding to one scan of the original source program or some internal representation of it. Machine dependent local and global optimizations are contained within the code generation and peephole optimization phases. These phases are divided into a specific pass structure. In particular, the lexical analysis, syntax analysis, and semantic analysis phases are combined into one pass. The machine independent local optimization is another pass; the machine independent global optimization and register allocation phases each consist of several passes. Code generation and assembly of object code are each one pass, while the peephole optimization may include several passes over the generated code.

To assist in understanding the compiler design a sample COL program is shown below. Whenever possible this program fragment is used as an example to illustrate the operation of each phase. It is the QUICKSORT example from section 7.6 of the COL Manual. Although this is only a fragment of a COL program, the COL compiler compiles each routine and function as it occurs. The whole module is considered a special routine, so this code fragment is indicative of the complete compiler operation.

```
// QUICK SORT routine WRITTEN IN COL
// TRANSLITERATED FROM
// ALGORITHMS + DATA STRUCTURES = PROGRAMS p. 80

declare ( ITEM is structure
                    ( KEY : integer
                      CONTENTS : integer))

routine QUICKSORT(ref A: array[1..?N] of ITEM)
    declare
        ( M = 12
            KEY : integer
            I,J,L,R : integer
            S : [0..M] initially 1
            STACK : array[1..M] of structure
                                    ( L : integer
                                      R : integer))

    STACK[1].L  := 1
    STACK[1].R := N
    repeat                          // TAKE TOP ENTRY FROM STACK
        L := STACK[S].L
        R := STACK[S].R
        S := S-1
        repeat
            I := L
            J := R
            KEY := A[(L+R)/2].KEY
            repeat
                while A[I].KEY < KEY do I := I+1 endwhile
                while KEY < A[J].KEY do J := J-1 endwhile
                if I <= J do
                    swap(A[I],A[J])
                    I := I+1
                    J := J-1
                endif
            until I > J
            if I < R do         // STACK REQUEST FOR RIGHT PARTITION
                S := S+1
                STACK[S].L := I
                STACK[S].R := R
            endif
            R := J
        until L >= R
    until S = 0
endroutine
```

Throughout the design document we refer to various algorithms and papers in the published literature, some of which are included as part of the COL design. Rather than repeat the details of these algorithms, we outline their operation and refer the reader to the original publication for the precise algorithm and the data structures required. If the paper is not directly applicable to the COL design as stated, we specify the modifications required.

The following two references are used throughout this document. Indeed, we assume that the reader is familiar with their contents.

[Gries]
David Gries, "Compiler Construction for Digital Computers", John Wiley & Sons, Inc., 1971

[Bauer]
F. L. Bauer, "Compiler Construction", Springer-Verlag, 1974

Chapter 2

Command Decoding

Command decoding is the the process of receiving from the operating system the user specification for this compilation. It is done separately from all other passes and consists of receiving from the operating system the input file name, output file names, and compilation options. This module must be operating system dependent. For example, on an IBM 370 the module would decode JCL commands while on TENEX the module would require a line from the user terminal which would decode into file names and options.

The output from the command decoding module is the input file identifier, listing file identifier, object file identifier, and a collection of internal compiler switches with associated data. The input file identifier is placed as the first entry in the macro invocation stack. Thus the input file is considered to be one large macro. When other files are needed by the "examine" directive they are pushed onto the macro stack above the input file (or any extant macro invocations). Thus the character handling routines have a uniform method for accessing characters from the input stream.

Chapter 3

Combined Lexical, Syntax, and Semantics Pass

The first pass of a COL compiler will consist of lexical analysis, syntax analysis, and semantic analysis of the input file. The input to the pass will be the macro stack containing the input file and a collection of options specified by the command decoder. These options may be boolean variables or boolean variables associated with other data. These three phases are handled as coroutines since the lexical analyzer and semantic analyzer will be called as subroutines from the syntax analyzer. The lexical analyzer will be called each time a new atom is needed, and the semantic analyzer called each time the instance of a rule has been identified and the grammar reduction is to be performed. The output of the pass will be six items:

Atom table of all atoms
Line table of line beginning positions
Block Structured Symbol Table
Flow Graph of Linearized Program
Linearized Program
Call Graph of function and routine calls

The atom table is the collection of all lexical units within the program. It contains one entry for each reserved word, operator, and identifier occurring in the program. There is only one occurrence for each identifier even though it may be declared

several t[illegible]oughout the rest of the compiler each lexical unit is identified by its index within the atom table. The line table is a list of entries, one for each line of text in the program. This entry contains the character number of the first character of this line and a pointer to a list of errors that have occurred in this line. It is used during the assembly phase to merge warning and error messages with the listing file. The block structured symbol table will contain entries for each defined identifier together with the attributes of that identifier. The flow graph will be a representation of the basic blocks together with predecessor and successor information. The linearized program will refer to identifiers by pointers into this symbol table. Other fields will later be filled in with predominator and postdominator information and loop structure information. The linearized program will have all scope structure removed, the scope being indicated by pointers into the symbol table where actual scope information will be implicitly or explicitly stored. Complex control structures will be translated into conditional branches and gotos. No definite decision has been made as to whether to store original control structure information, though for the moment we have decided against it. It can be added later, if needed, by indicating the start and end of the control structures with information-only entries in the

linearized program. The call graph of the program is a directed graph. There is one node for each routine or function in the program. There is one arc between two nodes if the predecessor node calls the successor node within the program. Currently the COL compiler will not use this data structure. However, current optimizing compiler papers are stressing the performance of interprocedure optimization. This will be useful in the future, but the current state of the art has not completely digested this approach so we have included the required data structures so that future implementations may be able to take advantage of interprocedure optimizations.

Chapter 4

Lexical Analysis

Lexical analysis is the division of the source file into small units which the rest of the compiler considers to be the indivisible portions of the program. These indivisible units include the single and double character operators, the reserved words, identifiers, and constants. We will refer to these indivisible portions as atoms. The lexical analyzer scans the source file of the program and emits a list of atoms. Actually the syntax analyzer calls the lexical analyzer as a co-routine and the lexical analyzer returns one atom each time called.

The lexical analyzer performs several major functions. It breaks the source file into atoms, inserts semicolons when the semicolon rule demands it, and expands macros when they are used. Its major function is the division of the source program into atoms. This is done by a finite state machine[Gries] as specified by the following grammar:

```
<token> ::= letter { letter ¦ digit ¦ _ }*
<token> ::= digit { digit ¦ _ }* {. digit { digit ¦
              _ }* ¦ } {{ E ¦ e } { + ¦ - ¦ }
            digit { digit ¦ _ }* ¦ }
<token> ::= { letter { letter ¦ digit ¦ _ }* # ¦ }
            $ { <char rep> ¦ " }
<token> ::= { letter { letter ¦ digit ¦ _ }* # ¦ }
            " { <char rep> }* "
<token> ::= digit { digit ¦ _ }* { # ¦ ! }
            { letter ¦ digit ¦ _ }+
```

```
<token> ::= ;
<token> ::= (
<token> ::= )
<token> ::= [
<token> ::= ]
<token> ::= ?
<token> ::= actual {
<token> ::= actual }
<token> ::= ,
<token> ::= @
<token> ::= +
<token> ::= -
<token> ::= /
<token> ::= { : ¦ > } { = ¦ }
<token> ::= < { = ¦ > ¦ }
<token> ::= . { . ¦ }
<token> ::= * { * ¦ = ¦ }
<token> ::= = { < ¦ > ¦ }
```

In this grammar, <token> is the root symbol for the grammar. The structure is basic BNF notation with the following additions. The metabrackets { ... } are used as BNF parentheses. The vertical bar ¦ indicates a choice of alternatives within metabrackets. If a metabracket is followed by a "*" then zero or more repetitions of the contents of the metabrackets are allowed. If the metabrackets are followed by "+" then one or more repetitions of the contents of the metabrackets are allowed. the notation "actual {" or "actual }" indicates that an actual { or } is allowed. The terminals "letter" or "digit" indicate any character of that form. This grammar is regular except for the non-terminal <char rep>, which cannot conveniently be described in BNF. It consists of those character constant forms described in section 5.1.3 of the COL Manual.

This non-terminal only occurs within character strings and character constants. The rest of this grammar can be implemented by finite state machines with a special escape when <char rep> is expected.

While performing its function the lexical analyzer maintains the atom table and line table. The atom table is actually two chain-linked hash tables. The first contains all reserved words and operators, the second contains all identifiers. When the lexical analyzer has found an atom, the atom is first search for in the reserved word portion of the atom table. If it is not found there it is searched for in the identifier atom table. If not found there it is inserted in the identifier atom table. Both of these tables are hash-coded for fast access and chain-linked so that atoms may be deleted on error recovery. Each entry in the atom table contains the following entries:

- Print name for the atom
- Pointers for maintaining hashing links
- Terminal number for syntax analyzer
- Can this terminal precede semicolon?
- Can this terminal follow semicolon?
- Linkage for alphabetizing the atom table

When single operators are found by the lexical analyzer they need not be retrieved from the hash tables. The compiler initialization stores a pointer to its atom table entry in a dedicated variable. Besides the reserved words and operators,

the quicksort example generates an atom table with the following identifiers:

ITEM	KEY	CONTENTS	QUICKSORT
A	N	M	KEY
I	J	L	R
S	STACK		

Note that the identifiers KEY, L, and R are used in two distinct contexts yet generate only one entry in the atom table. However, they will be recorded in two positions in the symbol table.

The line table is a simple list containing one entry for each line in the source file. This entry contains the position in the source file of the first character of this line and contains a pointer to the list of errors which occurred on this line.

Lexical analyzers are implemented by programming some form of finite state machines. The basic references are [Johnson] and [Joliat].

There are two basic methods of implementing these finite state machines. One consists of manually writing a routine which examines each character in turn and parses the atoms. The other method includes a small interpreter for finite state machines and a set of tables generated by a translator which specifies the specific machine. The first method is usually more efficient

while the second is more language independent. Since we are working with only one language this is not much of a restriction. Joliat's [Joliat] implementation of a lexical analyzer is of the second category but seems to be more efficient than the equivalent analyzer of the first type, at least for XPL. This efficiency results from his use of a matrix representation for character attributes rather than numeric comparisons to find them. His matrix representation is also more character set independent. We have chosen to structure the lexical analyzer as a combination of the two methods. It will be hand coded using matrix and vector representation of all character and atom attributes. This combination should maximize efficiency (since 30% or 40% of compile time will be in this module) and still maintain a large measure of machine independence.

The subroutines used by the lexical analyzer are Read_character and Write_character. Write_character will simply write a character to the listing file. Read_character will maintain two variables called This_character and Next_character indicating exactly what their names imply. The look ahead character will be needed for two-character atoms and comments.

The lexical analyzer will maintain a hash table of all atoms, discussed above. This table will contain the printing

name of the atom and all references to the atom outside the lexical analyzer will be by a pointer to this atom table. The table will not be used for scoping or control structures. It will be chain-linked for collisions with a fixed size hash area.

After the manually coded finite state machine has decoded the atom, it is looked up in the hash table. Reserved identifiers are stored there when the compiler is initialized. Simple atoms which can have only one meaning will not be looked up; instead the known pointer and value information are directly returned to the syntax analyzer. When an atom is an identifier it is looked for in the block structured symbol table. If found and if it is a macro identifier, the arguments that follow the identifier will be scanned. The scan will consist of looking for a left parenthesis, and then the first argument will consist of everything up to the first comma or right parenthesis, etc. The macro body will be added to the command stack together with the list of arguments. Read_character will then handle the substitution directly. Dummy arguments will be stored in the macro body as an ordered pair consisting of an illegal character followed by the ordinal number of the argument.

The lexical analyzer inserts semicolons in accordance with the semicolon rule in the COL Manual (section 5.5.3). This occurs

whenever a line feed, formfeed or vertical tab character follows one of the following lexemes:

> ID) enfunction endroutine label integer logical boolean interlock condition general float char endstart endfinish] end } endif endunless endtest endwhile endfor loop break retry stopswitch return endswitch endupon endregion endlock endfail fail endcode <integer> <floating number> <character constant> <character string> true false locked unlocked nil @

and precedes one of the following lexemes:

> ID examine public undeclare (function : routine forward declare check macro data open closed static dynamic location register selecton start finish private begin { if unless test while repeat for loop break retry stopswitch return switchon signal upon region lock unlock goto resultis failing fail code case default when assert explicit swap free

This language feature is implemented as follows. The lexical analyzer maintains three boolean variables. "Line_done" indicates that a line-ending character has been observed since the previous atom was formed. "Can_precede" indicates that the last atom formed could precede a semicolon. "Atom_ahead" indicates that the lexical analyzer is one atom ahead of the syntax analyzer. In that case the atom is stored in a variable called "Next_atom". When the lexical analyzer is entered it checks "Atom_ahead" to see if the atom required is already formed. If it is, "Next_Atom" is used and "Atom_ahead" is set false. If "Atom_ahead" was false the variable "Line_done" is set false, spacing characters are skipped and the atom is formed by the finite state machine. When

a line ending character is seen the variable "Line_done" is set true. Following the formation of the atom, if "Line_done" is true, "Can_precede" is true, and the formed atom can follow a semicolon, then save the formed atom in "Next_atom" and return a semicolon instead. If this boolean condition is not true return the formed atom after setting the variable "Can_precede" to indicate whether this atom can precede a semicolon.

Errors can be found during the atom forming process. They fall into one of the following classes:

. String or comment not terminated before end of file
. Illegal character beginning atom
. Badly formated number

In each of these cases an error flag is added to the line table for later inclusion in the listing. For errors of the first form, the terminating character is inserted and lexical analysis continues. For errors of the second form, the offending characters are deleted until a valid character for beginning an atom is found. For errors of the third form, the number is deleted and characters are skipped until a character which can begin an atom is found.

The lexical analyzer also decodes compiler directives and signals their occurrence to the rest of the compiler. The occurrence of the character "%" signals the beginning of a

compiler directive. The normal finite state machine for the lexical analyzer is suspended and a specialized finite state machine for compiler directives is invoked. Reserved words and operators no longer have their usual syntactic meaning. Instead the keywords for the directives and the form of each argument is decoded by this specialized finite state machine. Typically a compiler directive ends with a line terminator, although there are multiple line directives which have specialized terminators. Since not all the details of these directives are completely specified and some are object and source machine dependent, the directive finite state machine is not further specified at this point. A compiler directive which includes or is terminated by a line terminating character is considered to be a line terminating character for semicolon insertion.

In generating the bits strings for constants the lexical analyzer needs to be able to simulate the arithmetic of the object machine. This capability is also needed in the optimization phases of the compiler. Thus the compiler writer must supply a set of routines which will exactly simulate the arithmetic and logical instruction set of the object machine. This may not be possible with floating point arithmetic. In that case the routines must simulate the floating point hardware as best as possible. Whenever arithmetic is to be done on any

quantity which will be included in the object module, these routines must be used to perform it.

[Johnson]
W.L. Johnson, J.H. Porter, S.I. Ackley, and D.T. Ross, "Automatic Generation of Efficient Lexical Processors using Finite State Techniques", CACM vol. 11, p. 805-813, Dec. 1968

[Joliat]
M. L. Joliat, "On the Reduced Matrix Representation of LR(k) Parser Tables", University of Toronto Computer Systems Research Group Techn. Rep. CSRG-28, 1973

Chapter 5

Syntax Analyzer

The syntax analyzer groups the atoms generated by the lexical analyzer into expressions and statements. The standard ways to perform this task involve the use of one of the following techniques:

- recursive descent translation
- precedence parsers
- Floyd-Evans productions
- mixed precedence techniques
- LL(k) parsers
- LR(k) parsers

Recursive descent parsers are constructed by approaching the grammar as a top-down specification for a program. The program is generated as a set of recursive routines, one for each non-terminal. Although these parsers are the easiest to write, it is difficult to guarantee that the parser is consistent with the grammar. This problem is illustrated by subtle differences between the languages accepted by different PASCAL compilers. Precedence and mixed precedence parsers are table driven and the parser can be generated directly from the grammar. However, the set of grammars that these parsers will accept is limited. Floyd-Evans productions are a mechanism for generating recursive descent compilers. These productions provide a fine mechanism for error correction, but they suffer from the same lacks that the

recursive descent compilers suffer from. Certain kinds of precedence grammars admit to automatic generation of parsers using Floyd-Evans Productions. LL(k) and LR(k) parsers are table driven algorithms. The parser tables can be automatically generated from the syntax of the language by a generation program. Each of these methods has the benefit that errors in syntax are spotted at the first atom for which the preceding string of atoms is not the valid beginning of a program. LR(k) parser generators accept a more natural form of a language grammar than LL(k) parser generators. The tables for pure LR(k) parsers are two large to be practical, so we have chosen to use a variant of this approach called LALR(1). The tables for a LALR(1) tend to be smaller than precedence tables for large grammars and the parsing algorithm tends to be faster.

The LALR(1) parser algorithm is a state machine [Bauer]. The basic table for the parser is a table indexed by machine state and vocabulary symbol. The action of the parser is to read a vocabulary element from the lexical analyzer. The corresponding entry in the parser table indicates one of two actions. If the action indicated is a shift, the entry also indicates a destination state. The parser stacks the vocabulary element and the current state on a stack and enters the destination state. If the action indicated is a reduction, the entry also indicates a

grammar rule. The parser calls the semantic routine for this rule, removes the number of state entries and vocabulary elements from the stack indicated by the right hand side of the grammar rule, and enters the state corresponding to the state on top of the stack and the non-terminal left hand side of the grammar rule. These rules are repeated in turn until either a state is entered which has no transition for the vocabulary element read or reduction by the first grammar rule is indicated. If reduction by the first grammar rule is indicated, the end of source program has been reached and parser. The syntax phase is completed. If a state is entered with no transition for the current vocabulary element, then an error has been detected. A skeleton piece of COL program to implement this algorithm is as follows:

```
routine syntax()
    declare
      ( action : integer
        new_atom: atom)
    stack_ptr:=0                // initialize stack
    lexeme.atom_ptr:=nil
    lexeme.token:=eof_symbol
    char_ahead:=false
    atom_ahead:=false
    action:=start_state
    repeat
        test action < start_state do  // reduction
            new_lexeme:=semantics(action)  // call semantics phase
            stack_ptr:=stack_ptr-popnum(action)
            new_lexeme.token:=lside(action)
            action:=nstate(new_lexeme,state_stack[stack_ptr])
            stack_ptr:=stack_ptr+1
```

```
            symbol_stack[stack_ptr]:=new_lexeme
        otherwise
            stack_ptr:=stack_ptr+1
            symbol_stack[stack_ptr]:=lexeme
        endtest
        state_stack[stack_ptr]:=action
        if syntax_error(lexeme,action) do
            * announce error *
            * attempt spelling correction *
            * try local context error correction *
            * try simple error correction *
        endif
        action:=nstate(lexeme,action)
    until action=0
    * semantics for final reduction *
endroutine
```

In this representations of the algorithm the reduce states are stored as the rule number in the grammar. The shift states start after the reduce states. The start state is the first state after the grammar rules. The vocabulary element is stored in a variable called lexeme. The stack is implemented as two parallel arrays called Symbol_Stack and State_Stack. The routine popnum returns the number of entries on the right hand side of a rule. NSTATE is a routine which returns the entry in the two dimension array indicating the next state. Syntax_Error is a routine which indicates whether there is a valid transition in the parser table.

For error recovery this algorithm needs to be modified to provide a valid error correction method. Such a method most satisfy the following needs:

- allow continued parsering in most cases
- minimize the amount of source text skipped
- not introduce an infinite loop in the parser
- take actions to correct simple errors

The algorithms reported in [Morgan], [Feyock], [Leinius], [Peterson], [James], and [LaFrance] meet some of these needs. We have chosen the method of Feyock and Lazarus [Feyock], since it has the advantage of attempting local context changes to fix errors. The whole error correction scheme is as follows. First the error is reported. For each state in the LALR machine there is an error message, and this message is added to the error list for the current line number. Then spelling correction is attempted [Morgan]. Spelling correction is attempted only on reserved words and identifiers and spelling correction is only used if an identifier is changed to a reserved word. If spelling correction succeeds the parse is tried again. The main error correction method attempts to modify the input so as to make it valid. If this fails the analyzer will backup one atom and try again. A valid input must have both syntactic and semantic validity. (This is the reason for simultaneous syntax and semantic translation.) The backup procedure will go only as far as a language dependent point in the input called the beginning of a "substructure". This point in the COL is whenever a semicolon has just occurred or the stack has just popped elements

off the stack past the point on the stack at the beginning of this substructure. Atoms will be stored in a list from the beginning of a substructure until its end so that the backup can be done. Each atom will have stored with it in the symbol tables or linearized program the character number in the source of the first character of the atom. Whenever an error backup has occurred the tables of the compiler will be purged of all atoms which have associated numbers larger than the point of backup.

[Morgan]
Howard L. Morgan, "Spelling Correction in Systems Programs", CACM Vol. 13, No. 2, pp. 90-94, February 1970

[Feyock]
Stefan Feyock and Paul LaZarus, "Syntax-directed Correction of Syntax Errors", Software Practice and Experience, Vol. 6, pp. 207-219, 1976

[Leinius]
R. Leinius, "Error detection and Recovery for Syntax Directed Compiler Systems", Ph.D. Thesis, University of Wisconsin, 1970

[Peterson]
Thomas G. Peterson, "Syntax Error Detection, Correction, and Recovery in Parsers", Ph.D. Thesis, Stevens Institute of Technology 1972

[James]
Lewis R. James, "A Syntax Directed Error Recovery Method", Technical Report CSRG-13, Computer Systems Research Group, University of Toronto, Toronto, Canada, May 1972

[LaFrance]
Jacques LaFrance, "Optimization of Error Recovery in Syntax-Directed Parsing Algorithms", SIGPLAN Notices Vol. 5, No. 2, December 1970

Chapter 6

Semantic Analyzer

The semantic analysis phase forms each of the major internal forms of the source program from the form identified by the syntax analysis phase. It will consist of a set of subroutines. The support subroutines will add entries to the symbol table and begin or end scope regions. The graph subroutines will begin, end, and build basic blocks in the flow graph. The major subroutines will be the ones associated with grammar reductions. These will take the syntax information and add it to the three data structures.

The implementation of the semantic analysis phase is exactly like the implementation of the code generation phase of a one pass compiler such as PASCAL. For references on this approach to compiling see [Gries] and [Bauer]. We will summarize the following points which are peculiar to a COL compiler:

- How is the internal form generated
- Summary of internal form of expressions
- Summary of internal form of statements
- Processing of Declarations
- Type checking
- Summary of internal form for Flow Graph
- Summary of internal form for Call Graph
- Impact of Error recovery on Semantic Phase

The internal form is generated as a one pass compiler would generate code. However, the "machine code" is a special internal form for the COL compiler. When the syntax phase has determined that one of the rule reductions is required it calls the semantic phase. In particular, it calls the subroutine within the semantic phase for that particular rule reduction. The grammar specifying the COL syntax is as follows:

Grammar used for COL

```
<MODULE> ::= <MODULE HEAD> ; <SD-LIST>
<MODULE HEAD> ::= MODULE ID
<MODULE HEAD> ::= <MODULE HEAD> ; <MODULE HEAD ELEMENT>
<MODULE HEAD ELEMENT> ::= EXAMINE <ID-LIST>
<MODULE HEAD ELEMENT> ::= PUBLIC <ID-LIST>
<MODULE HEAD ELEMENT> ::= PUBLIC <ID-LIST> TO <ID-LIST>
<DECLARATION> ::= UNDECLARE ( <ID-LIST> )
<DECLARATION> ::= <MODE> FUNCTION ID <FPL> : <STORAGE> <TYPE> ;
     <START FUNCTION> <SD-LIST> ENDFUNCTION
<DECLARATION> ::= FORWARD FUNCTION ID <FPL> : <STORAGE> <TYPE>
<DECLARATION> ::= <MODE> ROUTINE ID <FPL> ; <START ROUTINE>
     <SD-LIST> ENDROUTINE
<DECLARATION> ::= FORWARD ROUTINE ID <FPL>
<DECLARATION> ::= DECLARE ( <DECL-LIST> )
<DECLARATION> ::= CHECK ID
<DECLARATION> ::= MACRO ID <MACRO INTERRUPT> =
<DECLARATION> ::= MACRO ID ( <ID-LIST> ) <MACRO INTERRUPT> =
<DECLARATION> ::= <STORAGE> DATA ( <ID-LIST> )
<DECLARATION> ::= <STORAGE> DATA ID ( <ID-LIST> )
<MODE> ::= <OPEN/CLOSED>
<MODE> ::=
<FPL> ::= ( <FP-LIST> )
<FPL> ::= ( )
<FP> ::= <CALL TYPE> <ID-LIST> : <STORAGE> <TYPE>
<CALL TYPE> ::= <VALUE/REF>
<CALL TYPE> ::= VARIADIC <CALL TYPE>
<CALL TYPE> ::=
```

```
<DECL> ::= <ID-LIST> : <VOLATILITY> <STORAGE> <TYPE> <INIT VALUE>
<DECL> ::= <ID-LIST> = E
<DECL> ::= <ID-LIST> = LABEL
<DECL> ::= <ID-LIST> IS <TYPE>
<DECL> ::= <ID-LIST> IS DIFFERENT <TYPE>
<VOLATILITY> ::= VOLATILE
<VOLATILITY> ::=
<STORAGE> ::= <STATIC/DYNAMIC>
<STORAGE> ::= <LOCATION/REGISTER> ( E )
<STORAGE> ::=
<INIT VALUE> ::= INITIALLY E
<INIT VALUE> ::=
<TYPE> ::= <SIZE> <UNSIZED TYPE>
<UNSIZED TYPE> ::= <BASIC TYPE>
<UNSIZED TYPE> ::= <AGGREGATE TYPE>
<UNSIZED TYPE> ::= FLOAT
<UNSIZED TYPE> ::= FLOAT ( E )
<UNSIZED TYPE> ::= <CHARSET> CHAR
<UNSIZED TYPE> ::= POINTER <TYPE>
<UNSIZED TYPE> ::= FUNCTION <FPL> : <STORAGE> <TYPE>
<UNSIZED TYPE> ::= ROUTINE <FPL>
<UNSIZED TYPE> ::= <DISCRETE TYPE>
<UNSIZED TYPE> ::= ID
<SIZE> ::= <INTEGER> <WORD/BYTE/BIT>
<SIZE> ::= <WORD/BYTE/BIT>
<SIZE> ::=
<CHARSET> ::= ID
<CHARSET> ::=
<AGGREGATE TYPE> ::= ARRAY <BOUND LIST> OF <TYPE>
<AGGREGATE TYPE> ::= STRUCTURE <S-MODE> <FIELD LIST>
<AGGREGATE TYPE> ::= <UN/PACKED/PARA> <AGGREGATE TYPE>
<ARRAY BOUND> ::= <DISCRETE TYPE>
<ARRAY BOUND> ::= ID
<FIELD LIST> ::= ( <FLD-LIST> )
<FIELD> ::= <S-MODE> ID : <VOLATILITY> <TYPE> <INIT VALUE>
<FIELD> ::= : <TYPE>
<FIELD> ::= SELECTON ID INTO ( <VAR-FIELD-LIST> )
<FIELD> ::= <S-MODE> <DECLARATION>
<FIELD> ::= <ASSERTION>
<FIELD> ::= START <SD-LIST> ENDSTART
<FIELD> ::= FINISH <SD-LIST> ENDFINISH
<VAR-FIELD> ::= <CASE LABEL> <FIELD LIST>
<CASE LABEL> ::= <CASE> :
<CASE LABEL> ::= <CASE> : <CASE LABEL> :
<S-MODE> ::= PUBLIC
<S-MODE> ::= PRIVATE
```

```
<S-MODE> ::=
<DISCRETE TYPE> ::= [ <LIMIT> .. <LIMIT> ]
<DISCRETE TYPE> ::= ( <ID-LIST> )
<LIMIT> ::= E
<LIMIT> ::= ? ID
S ::= <ASSERTION>
S ::= BEGIN <INIT SCOPE> <SD-LIST> END
S ::= %( <INIT SCOPE> <SD-LIST> %)
S ::= E := E
S ::= E =: <INFIX-OP> E
S ::= T10 ( <E-LIST> )
S ::= T10 ( )
S ::= IF E DO <INIT IF> <S-LIST> ENDIF
S ::= UNLESS E DO <INIT IF> <S-LIST> ENDUNLESS
S ::= TEST <INIT TEST> <ALT LIST> ENDTEST
S ::= TEST <INIT TEST> <ALT LIST> OTHERWISE <ADD OTHERWISE>
     <S-LIST> ENDTEST
S ::= WHILE E DO <INIT WHILE> <S-LIST> ENDWHILE
S ::= REPEAT <INIT REPEAT> <S-LIST> UNTIL E
S ::= FOR <FOR LIST ELEMENT> DO <INIT FOR> <S-LIST> ENDFOR
S ::= <ONE WORD S>
S ::= ID : S
S ::=
S ::= SWITCHON E INTO <INIT SWITCHON> <S-LIST> ENDSWITCH
S ::= <CASE> : S
S ::= UPON <ID-OR-LIST> LEAVE <INIT UPON> <S-LIST> DO <S-LIST>
     ENDUPON
S ::= SIGNAL ID
S ::= REGION E DO <INIT REGION> <S-LIST> ENDREGION
S ::= REGION E IFLOCKED <INIT REGION> <S-LIST> OTHERWISE <S-LIST>
     ENDREGION
S ::= LOCK E
S ::= LOCK E IFLOCKED <INIT LOCK> <S-LIST> ENDLOCK
S ::= <ONE ARG S> E
S ::= FAILING <INIT FAIL> <SD-LIST> FAILHERE <RESET FAIL>
     <SD-LIST> ENDFAIL
S ::= FAIL
S ::= FAIL FINISHING <E-LIST>
S ::= CODE ID DO <INIT CODE> <SD-LIST> ENDCODE
SD ::= <DECLARATION>
SD ::= S
<ALT LIST> ::= E DO <ADD ALT> <S-LIST>
<ALT LIST> ::= E DO <ADD ALT> <S-LIST> ORIF <ALT LIST>
<FOR LIST ELEMENT> ::= <FOR VARIABLE> := E STEP E UNTIL E
<FOR LIST ELEMENT> ::= <FOR VARIABLE> := E <INCR/DECR> E TO E
<FOR LIST ELEMENT> ::= <FOR VARIABLE> := E TO E
```

```
<FOR LIST ELEMENT> ::= <FOR VARIABLE> IN <DISCRETE TYPE>
<FOR VARIABLE> ::= ID
<FOR VARIABLE> ::= DEFINED ID
<CASE> ::= CASE E
<CASE> ::= CASE E TO E
<CASE> ::= DEFAULT
E ::= TO
E ::= WHEN E THEN E ELSE E
TO ::= T1
TO ::= TO EQV T1
T1 ::= T2
T1 ::= T1 OR T2
T1 ::= T1 <XOR/NOR> T2
T2 ::= T3
T2 ::= T2 AND T3
T3 ::= T4
T3 ::= NOT T4
T4 ::= T5
T4 ::= T5 <REL OP> T5
T4 ::= T5 = T5
T5 ::= T6
T5 ::= T5 <SHIFT OP> T6
T6 ::= T7
T6 ::= T6 <ADD OP> T7
T7 ::= T8
T7 ::= T7 <MUL OP> T8
T8 ::= T9
T8 ::= <ADD OP> T9
T9 ::= T10
T9 ::= T9 ** T10
T10 ::= T11
T10 ::= T10 ( <E-LIST> )
T10 ::= T10 ( )
T10 ::= T10 [ <E-LIST> ]
T10 ::= T10 . ID
T10 ::= T10 @
T10 ::= <CONVERT/FORCE> ( <TYPE> : E )
T10 ::= <CONSTRUCT/ALLOCATE> ( <AGGREGATE VALUE> )
T10 ::= TABLE ( <TYPE> , <E-LIST> )
T11 ::= ( E )
T11 ::= ID
T11 ::= <INTEGER>
T11 ::= <CONSTANT>
<AGGREGATE VALUE> ::= <TYPE> , <FIELD VALUE LIST>
<AGGREGATE VALUE> ::= <TYPE> : E
<AGGREGATE VALUE> ::= <TYPE>
```

```
<FIELD VALUE> ::= <FIELD LABEL LIST> : E
<FIELD VALUE> ::= <FIELD LABEL LIST> ( <FIELD VALUE LIST> )
<FIELD LABEL LIST> ::= <FIELD LABEL SEQUENCE>
<FIELD LABEL LIST> ::= <SUBSCRIPT LIST> . <FIELD LABEL SEQUENCE>
<FIELD LABEL LIST> ::= <SUBSCRIPT LIST>
<FIELD LABEL SEQUENCE> ::= <FIELD LABEL>
<FIELD LABEL SEQUENCE> ::= <FIELD LABEL> . <FIELD LABEL SEQUENCE>
<FIELD LABEL> ::= ID
<FIELD LABEL> ::= ID <SUBSCRIPT LIST>
<SUBSCRIPT LIST> ::= <SUBSCRIPT ITEM>
<SUBSCRIPT LIST> ::= <SUBsCRIPT ITEM> <SUBSCRIPT LIST>
<SUBSCRIPT ITEM> ::= [ <SUBSCRIPT SEQUENCE> ]
<SUBSCRIPT SEQUENCE> ::= <SUBSCRIPT>
<SUBSCRIPT SEQUENCE> ::= <SUBSCRIPT> , <SUBSCRIPT SEQUENCE>
<SUBSCRIPT> ::= E
<SUBSCRIPT> ::= E .. E
<ASSERTION> ::= ASSERT E
<S-LIST> ::= S
<S-LIST> ::= S ; <S-LIST>
<SD-LIST> ::= SD
<SD-LIST> ::= SD ; <SD-LIST>
<E-LIST> ::= E
<E-LIST> ::= E , <E-LIST>
<ID-LIST> ::= ID
<ID-LIST> ::= ID , <ID-LIST>
<FP-LIST> ::= <FP>
<FP-LIST> ::= <FP> , <FP-LIST>
<DECL-LIST> ::= <DECL>
<DECL-LIST> ::= <DECL> ; <DECL-LIST>
<BOUND LIST> ::= <ARRAY BOUND>
<BOUND LIST> ::= <ARRAY BOUND> , <BOUND LIST>
<ID-OR-LIST> ::= ID
<ID-OR-LIST> ::= ID OR <ID-OR-LIST>
<FIELD VALUE LIST> ::= <FIELD VALUE>
<FIELD VALUE LIST> ::= <FIELD VALUE> , <FIELD VALUE LIST>
<FLD-LIST> ::= <FIELD>
<FLD-LIST> ::= <FIELD> ; <FLD-LIST>
<VAR-FIELD-LIST> ::= <VAR-FIELD>
<VAR-FIELD-LIST> ::= <VAR-FIELD> ; <VAR-FIELD-LIST>
<INFIX-OP> ::= EQV
<INFIX-OP> ::= OR
<INFIX-OP> ::= <XOR/NOR>
<INFIX-OP> ::= AND
<INFIX-OP> ::= <REL OP>
<INFIX-OP> ::= =
<INFIX-OP> ::= <ADD OP>
```

<INFIX-OP> ::= <MUL OP>
<INFIX-OP> ::= **
<MACRO INTERRUPT> ::=
<START FUNCTION> ::=
<START ROUTINE> ::=
<INIT IF> ::=
<INIT WHILE> ::=
<INIT REPEAT> ::=
<INIT FOR> ::=
<INIT SWITCHON> ::=
<INIT UPON> ::=
<INIT REGION> ::=
<INIT LOCK> ::=
<INIT FAIL> ::=
<RESET FAIL> ::=
<INIT CODE> ::=
<INIT SCOPE> ::=
<INIT TEST> ::=
<ADD OTHERWISE> ::=
<ADD ALT> ::=

Terminals for COL Grammar

MODULE	ID	;
EXAMINE	PUBLIC	TO
UNDECLARE	(	)
FUNCTION	:	ENDFUNCTION
ROUTINE	ENDROUTINE	FORWARD
DECLARE	CHECK	MACRO
=	DATA	<OPEN/CLOSED>
<VALUE/REF>	VARIADIC	LABEL
IS	DIFFERENT	<STATIC/DYNAMIC>
<LOCATION/REGISTER>	INITIALLY	<BASIC TYPE>
FLOAT	CHAR	POINTER
<WORD/BYTE/BIT>	ARRAY	OF
STRUCTURE	<UN/PACKED/PARA>	SELECTON
INTO	START	ENDSTART
FINISH	ENDFINISH	PRIVATE
[	]	..
?	BEGIN	END
{	}	:=
=:	IF	DO
ENDIF	UNLESS	ENDUNLESS
TEST	ENDTEST	OTHERWISE
WHILE	ENDWHILE	REPEAT
UNTIL	FOR	ENDFOR
<ONE WORD S>	SWITCHON	ENDSWITCH
UPON	LEAVE	ENDUPON
SIGNAL	REGION	DO
ENDREGION	IFLOCKED	OTHERWISE
LOCK	ENDLOCK	<ONE ARG S>
FAILING	FAILHERE	ENDFAIL
FAIL	FINISHING	CODE
ENDCODE	ORIF	STEP
<INCR/DECR>	IN	DEFINED
CASE	DEFAULT	WHEN
THEN	ELSE	EQV
OR	<XOR/NOR>	AND
NOT	<REL OP>	<SHIFT OP>
<ADD OP>	<MUL OP>	**
<CONVERT/FORCE>	<CONSTRUCT/ALLOCATE>	TABLE
<INTEGER>	<CONSTANT>	ASSERT
,	.	@
VOLATILE		

This grammar is automatically generated from the language specification document. That is, a by-product of processing the COL Manual by the text formatter is a file containing the syntax. (Thus changing the syntax in the manual results in the compiler's changing also.) This syntax is then massaged in various ways to decrease the size of the parser tables. First, the following non-terminals are eliminated as unnecessary:

<data declaration>	<function declaration>
<routine declaration>	<scalar declaration>
<event declaration>	<macro declaration>
<undeclaration>	<variable decl>
<constant decl>	<type decl>
<other type>	<simple statement>
<conditional statement>	<iteration statement>
<interlock statement>	<other statement>
<machine-like code>	<sequence>
NL	NC
<primary expression>	<operator expression>
<aggregate expression>	<aggregate init>
<other expression>	<macro param>
<module ID>	<computer ID>
<failure statement>	<switch statement>
<Zahn statement>	<event>
<memory descriptor>	

Certain of the terminals are then grouped together into a single terminal since they all occur in the same contexts:

New Terminal	Old Terminals
<OPEN/CLOSED>	OPEN CLOSED
<VALUE/REF>	VALUE REF
<STATIC/DYNAMIC>	STATIC DYNAMIC
<WORD/BYTE/BIT>	WORD BYTE BIT
<ONE WORD S>	LOOP BREAK RETRY STOPSWITCH RETURN
<ONE ARG S>	UNLOCK GOTO RESULTIS
<CONVERT/FORCE>	CONVERT FORCE
<CONSTRUCT/ALLOCATE>	CONSTRUCT ALLOCATE
<CONSTANT>	<FLOATING NUMBER>
	<CHARACTER CONSTANT>
	<CHARACTER STRING>
	<LOGICAL CONSTANT>
	TRUE FALSE NIL LOCKED UNLOCKED

Furthermore, <basic type> is changed to a terminal including the simple types but excluding FLOAT and CHAR. Type GENERAL is included in <basic type>. The following built-in functions and routines are handled as functions and routines by the syntax analyzer:

MAX	MIN
ABS	TRUNCATE
ROUND	FLOOR
CEILING	EXPLICIT
HIGH	LOW
SWAP	FREE

Finally, the precedence rules for arithmetic are included in the syntax to remove the ambiguity in the grammar. To implement one pass compilation of internal code, empty reductions are included to force the call of semantic routines at critical places in the recognition of a rule. For example, in a block a semantic routine must be called immediately after the BEGIN so that the new scope for the symbol table can be started.

Expressions are stored internally as trees, as this form is more easily optimized. Many of the local optimizations performed in the local optimization phase can be considered pattern matching problems on trees. The tree form also helps the code generation and register allocation phases. It hinders the global optimization phase only slightly. The structure of the tree node is as follows:

- operator
- pointer to list of arguments
- type of result
- pointer to next argument in this argument list

The arguments of an operator are stored as a list rather than a binary tree. This form aids in handling associative operators while causing only a small space penalty for non-associative operators. The operators include the standard operators of the language, as follows:

left_shift	right_shift	left_rotate
right_rotate	less_than	greater_or_equal
greater_than	less_or_equal	equal
not_equal	and_operator	or_operator
eqv_operator	nor_operator	xor_operator
plus_operator	minus_operator	multiply_operator
divide_operator	mod_operator	power_operator
unary_minus	not_operator	index_operator
select_operator	val_operator	explicit_operator
function_call	convert_operator	force_operator
when_operator	truncate_operator	round_operator
floor_operator	ceiling_operator	table_operator
construct_operator	allocate_operator	

The operator field of the tree is in the form of an enumeration data type with these field names. An expression is a pointer to

one of three kinds of nodes: an expression node as specified above, an entry in the symbol table for a variable or field reference, or an entry in the constant table for a constant reference.

The statements in internal form are stored as a doubly-linked list, aiding in the optimization phases where forward and backward scans over the statements are made. There are two kinds of statements. The first is the primitive form of COL statement. For example, the two forms of assignment statements, the conditional statements, and the goto statement. The more complex COL statements are translated into these statements so that a uniform optimization technique may be used. In the absence of a "goto" statement this reduction would not be necessary. The BLISS-11 compiler performs well using only a tree structure form of the statements. However, the "goto" statement destroys the tree structure of statement execution. With this limitation we have chosen to return to the flow graph with basic block form of statement representation for optimization. The primitive statement forms are:

E1 := E2	Assignment
E1 *= <binary op> E2	Self Modification
Claim E	Compiler generated information
E1(E2,...)	Procedure Call
if E goto L	Conditional Statements
if not E goto L	
if E = 0 goto L	

```
if E <> 0 goto L
if E = locked goto L
if E <> locked goto L
if E = nil goto L
if E <> nil goto L
if E1 > E2 goto L
if E1 <= E2 goto L
if E1 < E2 goto L
if E1 >= E2 goto L
on E goto (L1,...)          Jump Table
return                      Return from routine
resultis(E)                 Return from function
lock E iflocked goto L      Primitive Lock statement
unlock E                    Primitive unlock statement
assert E iffalse E          Used for run-time checks
finalize(E)                 Encapsulated Type Finalization
initialize(E)               Encapsulated Type initialization
```

The internal representation of a statement is as a structure containing one field to indicate the kind of statement and one field for each expression or label required. The semantic routines will choose that form of statement which encodes the most information at the statement level. For example, the semantic routines will choose that representation of a conditional statment which involves the relational operators when a relational operator occurs at the top level.

The internal form of the program does not include any declarations. The declaration information is stored in the symbol table. Associated with the symbol table entry are pointers to the beginning and end of the scope of the declaration. Declarations are processed as in the usual one-pass compiler with the

following two exceptions. First, closed function and routine declarations are processed to completion before the syntax and semantic phases continue. Thus at the completion of the recognition of a function or routine declaration the remaining phases of the compiler are performed on the body of the function or routine. This technique limits the size of internal data tables necessary to perform optimizaton. Should interprocedure optimization be added this design will have to be reconsidered. Secondly, the "declare" construction requires multiple passes over the declaration to obtain all information. During the parsing of the declaration a tree structure is built specifying the structure of the declaration. During this pass the structure of each object in the declaration is determined. Those identifiers which will be types are noticed and the basic structure of each object is determined. After the first pass the various data structures for the identifiers in the declaration are known, but there are fields to be filled in. The semantic routine now makes multiple passes over these structures attempting to fill in information determined in the previous pass. This process stops whenever all fields are filled in, indicating successful processing of the declaration, or no fields were filled in on the last pass, in which case an error has been detected.

During the generation of the internal program form, the "type" of each identifier, constant, and expression is checked against the type of expression required in the immediate context. If they do not agree an error is signaled. Types are handled in the COL exactly as in the PASCAL compiler [Wirth]. There are two exceptions. In PASCAL a type declaration creates a new type while in COL a type declaration is an abbreviation for the old type. In PASCAL a subrange type is distinct from the base type while in the COL a subrange type is the same as the base type but with extra information provided. To speed the processing of type comparisons all types are stored in a hash table. The hash function is a function of the base type and any fields that occur in the type declaration. Each type, when defined, is searched for in this table and entered if not found. When a type with the "different" attribute is generated, it is automatically added to the hash table. In this fashion the compiler keeps only one copy of each type. Hence type checking can be performed by a simple comparison of pointers in most cases.

The flow graph records the possible control flow of the program. The internal form of the program is divided into basic blocks, each beginning at a label and continuing until a conditional statement, a goto, or another label occurs. The basic blocks are the nodes of the flow graph. There is an arc in

the flow graph between any two basic blocks such that the first can directly transfer to the second. The internal form of the flow graph is a structure in the free storage area. Each node in the flow graph contains:

. pointer to beginning of basic block
. pointer to end of basic blcok
. pointer to list of successor blocks
. pointer to list of predecessor blocks
. opiminzation fields for later use.

The call graph is a representation of program flow between subroutines, and is included for later introduction of interprocedure optimization. There is one node for each routine or function, with an arc between nodes if the predecessor node can call the function or routine associated with the successor node. There will be various fields within each node and arc. We do not now specify them since they depend upon the choice of interprocedure optimization techniques chosen.

The semantic routines are directly involved in error detection and recovery. Whenever a semantic error is detected the semantic routine returns to the syntax phase a flag indicating error. This flag is used by the syntax phase when it is in error recovery mode to determine the correction to perform for continuing processing. One of the requirements of the error recovery method chosen is that all additions to data structures

in the semantics phase must be reversible. This is done by delaying until the local optimization phase those transformations (such as short circuit evaluation) which dramatically change the structure of the internal form of the program. With each data structure entry in the semantic phase is stored a sequence number to indicate the position of the atom associated with this data in the source file. When an error is detected, the syntax phase may backup the semantic and syntax processing by purging all tables and structures of entries with later sequence numbers than the point to which we wish to return.

[Wirth]
N. Wirth, "The design of a PASCAL compiler", Software -- Practice and Experience, 1 (1971) pp 309-333.

Chapter 7

Local Machine Independent Code Optimization

This phase of the compiler provides the following functions. First, it changes invocations of open functions and routines into in-line code, and then it simplifies expressions, making complicated expressions less complicated. Finally, it changes complex boolean expressions into simpler boolean expressions using the short circuit mode of evaluation. The input to this phase is the linear code text from the semantic phase, the symbol table, and the flow graph. The flow graph is used only in the sense that it is modified by the short circuit and open procedure expansion.

The local optimization module looks at the linearized code form of a function or a routine. It performs a linear pass over this code form analyzing each statement and each expression within a statement. Each expression will be replaced (if possible) by another expression which evaluates to the same value but uses fewer instructions or registers. This replacement is made by transforming the tree representing the expression into a simpler tree using only properties that are dependent on the local context of the expression. These contexts can be stated in a machine independent way. However, determining which

transformations are an improvement is machine dependent. Except for object machine arithmetic in the lexical analyzer this is the first machine dependent portion of a major phase of the compiler. To minimize the machine dependence of this module the following organization will be used. To determine if the code transformation is an improvement, one must know some information about the structure of the destination machine. For example, if the machine has load negative and store negative opcodes, the planned compiler will perform complex minus removal. However, if the machine has only a load negative and no store negative, minus removal is not always possible. The situation becomes even more restricted if neither a load negative nor a store negative instruction is available. Though optimizations may be machine dependent, many of the optimizations can be stated in a machine independent form. We propose to provide flexibility in optimization by incorporating parameterization and object functions into the local optimization phase. Each optimization will have a parameter associated with it. This parameter will specify whether the particular optimization should be attempted. In cases similar to the minus problem, several parameters will be given to specify which subsections of the algorithm can be useful. These parameters are specified by the system designer for the destination machine when the compiler is created. Hence,

only appropriate optimizations will be applied to code for a particular machine.

Some optimizations cannot be easily controlled. For example, common subexpression optimization can be useful for some expressions and not useful for others. This discrepancy cannot be handled by a single parameter; instead, we propose that an object function for such optimizations be provided. This object function will receive data about the proposed optimization as arguments and return an answer. The answer will either be a simple yes or no, or some sizing information which the general optimization algorithm can use to determine the value of the proposed action. In the common subexpression case, the object function could return an approximation to the size of code for computing this expression. The size function will be given the pointer to the tree for the expression and will use information about the destination machine to return the size and words or bytes of the code.

Local optimizations are those transformations to the program that take place within a single statement. The local optimizations include:

. local common subexpression optimization

. unary operator simplification

. the de Morgan's rule applied to boolean expressions

. replacing multiplications by two by shifts

There is a full list of such optimizations in [Bagwell] and [Fraley]; we propose that these optimizations be included in the COL compiler. Further local common subexpression elimination, constant folding, and variable subsumption can be accomplished using the value-number algorithm presented in [Cocke]. We propose that this algorithm be concluded for each straight line section of code, called a basic block. According to [Lee], these optimizations can provide about 50% of the code improvement that could be performed by optimization. Furthermore, these optimizations can all be done in time proportional to the length of the program.

All local optimizations will be performed in this one phase. In one approach to compiler writing, some optimizations are performed in the semantic phase of the compiler. We do not do this since error recovery requires that all actions taken in the semantic phase be reversible. Local optimizations will always be performed. This is not always an obvious choice since certain of the local optimizations might destroy obvious common subexpressions. The alternative in making this choice seems to be the inclusion of combinatorial algorithms whose cost would be unacceptable in an operational compiler.

The local optimization phase of the compiler accepts its input, the linearized code, the symbol table and the flow graph of the program. These are also the outputs from this phase. It creates no new data structures, but modifies these structures to indicate an optimization. Each expression is replaced by the best expression that can be found with equivalent semantics. These expressions can be determined by the value-numbering algorithm [Cocke] and tree modifications based on the rules of algebra. The value numbering algorithm can be adapted with the following modifications. First, we have included the COL's "when" expression in our internal linearized form, although doing so violates the usual view that the basic block is a section of code with only one execution path through it. We may have an execution path which is a simplified form of a cyclic directed graph. However, on the branches of the cyclic graph variables may not be changed but only evaluated. Thus, a modified form of the extended basic block algorithm, as proposed in [Marateck], may be used to determine common subexpressions even in this case. Furthermore, the whole Kennedy algorithm will be applied to extended basic blocks to provide constant elimination and variable subsumption on a wider scale than in the purely linearized tests.

To determine the best form of an expression, first the following optimizations will be applied to the expression to simplify them.

1. Eliminate unnecessary operations
   - I*0 replaced by 0
   - I+0 replaced by I
   - I*1 replaced by I
   - A**1 replaced by A
   - I/1 replaced by I
   - I-I replaced by 0
   - -(-A) replaced by A
   - A+0 replaced by A if A normalized
   - B and TRUE replaced by B
   - B and FALSE replaced by FALSE
   - B or TRUE replaced by TRUE
   - B or FALSE replaced by FALSE
   - B eqv TRUE replaced by B
   - B eqv FALSE replaced by not B
   - B xor TRUE replaced by not B
   - B xor FALSE replaced by B
   - B and B replaced by B
   - B and not B replaced by FALSE
   - not not B replaced by B

2. Floating point optimizations
   - A/constant replaced by A*(1/constant) if exact
   - A**(-constant) to 1/A**constant if small constant
   - A**constant replaced by multiplies for small constants
   - A/2 or A*.5 replaced by half operation if it exists
   - 2*A replaced by A+A
   - 4*A replaced by (A+A)+(A+A)

3. Integer optimizations
   - (-1)**I replaced by odd/even test
   - I*2**constant replaced by shift
   - I*(2**k+2**j) replaced by shifts and adds
   - I*(2**K-2**J) replaced by shifts and subtracts if I small

4. General optimizations
   - Eliminate common subexpressions
   - Replace variable by known constant value
   - Replace variable by another variable if value same
   - Eliminate assignment if known to not change value

Eliminate assignment if value not used before reassigned
Use distributivity on subscripts to group constants

Then the expression will be looked up in the available expression table. This table is indexed by operators and value numbers and contains all expressions which are available at the current point in the program. If a better expression can be found, namely a constant, or a variable which has the same value as does the expression, the expression will be replaced by the constant or the variable. If a constant occurs in the expression, the context the expression occurs in will be investigated to see if simplifications of program, elimination of code, or the changing of conditional to unconditional branches can occur. After this, the expression itself is inserted into the available expression table to be used in future common subexpression elimination. Finally, at the end of each basic block, the compiler will trace back to see if there are multiple assignments to one variable. If there are multiple assignments to a variable, with no uses in between, the first assignment will be eliminated. This can be caused by constant subsumption, even if the programmer does not have multiple assignments to a variable.

There is one further modification to the value number scheme. Schwartz's statement of the value number scheme for common subexpression elimination does not consider elements in

free storage or fields of a structure. Elements in free storage can be handled simply by treating them as elements of an array. In particular, all elements in free storage of the same type can be considered to be elements of an array, indexed by the pointers to that particular item. Then the value number of Schwartz will work with elements in free storage. We must only consider that there is one dummy element which is an array of all elements in free storage of a particular type. Elements in a structure are a bit more difficult to handle. If each element of a structure is considered to be a separate variable with its own value numbers, then the evaluation of common subexpression poses no problem. However, if there is an assignment to the whole structure the value number of each element in the structure must change. Hence, an element in a structure should be looked upon as an expression with one argument being the field, with its value number and the other argument being the value number of the structure. Thus, if a copy to the whole structure takes place, the value number of the structure changes and each field can no longer be involved in common subexpressions. However, if there is assignment to only one field, the whole structure has indeed changed, but all non-changed fields are still available. Thus, when assignment to a particular field of a structure occurs, the value number of the structure changes, the value number of that

field changes, but each element of the structure except this one should be added to the available expression stack for availability in common subexpression elimination just as when a store to a particular element of an array changes the value number of the array, but that particular element is made available as a special case. With these modifications, the Schwartz and Kennedy number scheme will work for the COL and provide good local optimization for expressions.

During this linear scan through the code text, the occurrences of open routines and open functions are noticed. When an open routine occurs, it can be handled in one of several ways. In each of these cases, the code is inserted in line in the code text. The special cases involve how the arguments to the routine are processed. For each argument, if it or any of its constituent parts are not changed within the routine body, the actual parameter can replace each occurrence of the corresponding formal parameter in the code body. However, if one of the parts of the actual parameter is changed, then the expression must be evaluated before entry into the body of the routine and the expression then referred to by a pointer. This special kind of pointer, which has been included in the internal form of expression, involves both a type and an expression tree. The expression tree is the value the pointer points at. For

example, if A[I] is an actual parameter to an open routine, the expression tree for A[I] is passed along with the pointer. Within the body of the routine the compiler will treat the actual parameter as the expression represented by the expression tree until such time as one of the variables in the expression tree changes, at which time the value numbers have to be changed, so that the value is guaranteed not to exist for common subexpressions elimination and for code generation. Similar problems occur with open functions, but there is one more dilemma: The function can occur deep within an expression. In a prefix walk through the expression tree, when a function occurs, we attempt to reorganize the expression to minimize the number of temporaries that will have to be saved. After having done this, we save all expressions which have to be evaluated before the evaluation of this function in temporaries, declared to have appropriate scope, and then we assign the value of the function to a temporary. At this point, we can open the function into in-line code, replacing all occurrences of resultis commands by assignments to this particular temporary.

This procedure will eliminate all open routines and functions. For those routines and functions that have not been declared either open or closed, the compiler is at liberty to choose for itself which method to use. The way the choice will

be made is for a machine dependent routine to be written which estimates the size of the code. During the semantic and syntax phases, the number of occurrences of this procedure are noted, and a heuristic approximation of the amount of code necessary for open and for closed occurrences is evaluated. The smaller choice will be used. The choice is affected by a request by the programmer to optimize space or time.

[Bagwell]
John T. Bagwel, Jr., "Local optimizations", SIGPLAN Notices Vol 5, No 7, Jul 1970, pp 62-66.

[Fraley]
Dennis J. Fraley, "Expression optimization using unary complement operators", SIGPLAN Notices Vol 5, No 7, Jul 1970, pp 67-85.

[Cocke]
John Cocke and J. T. Schwartz, "Programming languages and their compilers", Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, N.Y., Apr 1970.

[Lee]
John A. N. Lee, "The Anatomy of a Compiler", second edition, Van Nostrand Reinhold Company, New York, 1974.

[Marateck]
S. Marateck and J. Schwartz, "A SETL Program for a Basic Block Optimizer and Extended Basic Block Optimizer", SETL Newsletter #105, Courant Institute of Mathematical Sciences, New York University, New York, N. Y., April 1973

Chapter 8

Global Optimization Phase

Global optimizations are those transformations involving enough of the program so that there is more than one execution path through the code segment in question. The following global optimizations will be performed:

- common subexpression elimination
- code motion out of loops
- strength reduction within loops
- global constant elimination

We propose that the COL compiler implement these optimizations by the following algorithms. Global constant propagation will be implemented using the p-graph algorithm proposed by Massachusetts Computer Associates. The other optimizations will be implemented using the interval analysis method proposed by Schwartz, Cocke, and Allen.

One frequently used optimization is constant subsumption. It involves replacing the currents of a variable X by a constant C when the latest assignment to X was of the form of X:=C. This is a useful optimization for several reasons. First, programmers with FORTRAN experience write programs in which parameterization is done by such assignments. Granted we have provided manifest constants for such purpose; however, history shows that

programming habit is well nigh incurable. Secondly, a programmer may prefer to make such an assignment and then initialize the data associated with it. For example, the programmer might wish to initialize an index to an array and then initialize all the parallel data arrays. He might prefer to use the index rather than the constant since all references to these arrays use that index. Finally, open routines can introduce many of these constructs when the subroutine is expanded.

There are several ways to implement this optimization, depending on the scope of the application and the algorithm used. First, it is easy to perform Cocke's subsumption in straight line code. The compiler need only remember those variables that have been set to constants and not changed. This is the method often used in compilers that do not do constant subsumption. Secondly, the compiler can notice those variables which are set only once and used several times. If the assignment to the variable is a constant, then all its uses can be replaced by that constant. The compiler should be careful to notice if all uses of the variable are reachable from the assignment. If they are not, the compiler should generate a warning message; otherwise, the compiler will have eliminated an obvious error condition. Third, the p-graph algorithm proposed by Massachusetts Computer Associates can be used to check global flow and assignment. The

algorithm divides the program into regions where one and only one assignment has effect. If there is a merging of control flow, a pseudo-assignment is generated. The compiler can scan the assignments for those of the form X:=C and replace all uses of that assignment by that constant. If there are no pseudo-assignments reachable from this assignment, the assignment can be eliminated. Finally, Kildall in [Kildall] proposed an algorithm for constant subsumption detection.

We propose the following strategy. First, the straight line algorithm will be applied to each section of straight line code in the local optimization phase. Next, those variables which have only one assignment which is to a constant will be eliminated and the constant will replace all occurrences. Finally, the p-graph algorithm will be applied to those variables which are at some point set equal to a constant and the constant used outside the one straight line sequence of code.

This strategy has been chosen for the following reasons. Although we need global constant subsumption to gain the maximum benefit from open subroutines, it is expensive in space and time to perform. Hence we will handle the special cases first. There are two special cases which are straight line in the one assignment. We choose the p-graph algorithm over Kildall since

Kildall must be applied to all the variables at once while the p-graph can be computed for only those variables we need. The constant subsumption algorithm will be applied before other global flow analysis algorithms since the introduction of constants will simplify the rest of the program and will improve its chances of finding loop invariant expressions within loops.

The other optimization transformations could be implemented in several ways. Earnest in [Earnest] has proposed one set of algorithms for performing all the global optimizations except strength reduction. They were implemented in yet another way in the FORTRAN level H compiler, and Cocke and Allen and Schwartz have proposed the interval analysis method. In all three of these methods the varied information is stated as a set of equations which must be solved, and each performs the solution in a different fashion. The method with the most literature associated with it and with the most explicit algorithms stated is the interval analysis method of Cocke and Allen and Schwartz. Its one difficulty is that it does not work with all programs without a modification. That modification involves changing the flow graph of the program if a certain anomalous program structure exists. This program structure cannot exist in structured programs; it can only exist with multiple entry loops, and we therefore propose to implement this algorithm with special

handling for this anomalous case. Our global analysis phase will follow closely the analysis phase proposed by Shaefer[Schaefer].

Following constant elimination, the global optimization phase of the compiler will perform the following optimizations:

. redundant subexpression elimination
. code motion out of frequently executed regions
. strength reduction
. linear test elimination

These techniques have been approached in several ways by Earnest and Cocke and Schwartz. Various regions have been proposed for the basis of these algorithms ranging from intervals to general strongly connected regions. Furthermore, several methods for solving the equations involved have been proposed. An iterative approach have been used by Ullman [Ullman], the interval analysis approach by Allen [Allen-72], and recently more sophisticated elimination algorithms have been proposed by Graham [Graham] and Tarjan [Tarjan]. We have chosen the interval analysis method for solution of these algorithms for the following reasons. First, these seem to be the best documented methods available in the literature; they provide both efficient implementation and implementation in all situations. The algorithms are documented in Cocke and Schwartz [Cocke-70], the SETL newsletters [Kennedy-71 through Kennedy-74a], and Schaefer [Schaefer]. The iterative analysis method is competitive but does not seem to

have a great advantage (see [Kennedy-76]). The more recent methods by Graham and Tarjan, although faster in the limit, seem to involve large amounts of computation to begin the algorithms and large constants of proportionality. However, the method of solution can be changed in the future.

The interval analysis method reduces these two problems. First, some programs are not reducible. About 90% of programs are reducible, and certainly all structured programs are reducible. However, some programs with "goto"s will not be reducible. We propose to optimize within the program until the reductions involved lead to irreducible graphs. Each irreducible graph will be handled as a loop for optimization puposes. Hence at that point all code motion must be out of the whole subprogram or it will not occur. Irreducible graphs can only occur if the program contains multiple entry loops. The second problem involves the elimination of redundant computation. Interval analysis will provide a function which will declare that a particular instance of a computation is unnecessary. We propose to eliminate all such unnecessary computations. Interval analysis will then provide a function which will indicate whether an expression is used at some point following any given point of computation. We will use this function to guarantee that each instance of an expression which is used will be stored. This

will eliminate unnecessary computations, but may possibly introduce a store into a loop. This will probably occur infrequently since most likely the expression will be stored into a register on a multiple register machine and thus cost nothing. If the loop is sufficiently complex that a register cannot be used, then the loop is probably large enough that the store will not significantly affect efficiency in any case.

The structure of the implementation of the global optimization scheme using interval analysis follows very closely the structure suggested in [Schaefer]. However, we have modified it to include the simplifications suggested in the discussion of the interval analysis algorithms in Schwartz's monograph [Schwartz]. In particular, Schaefer is extremely cautious about the motion of code and about when code motion is profitable. Schwartz uses heuristics to determine when code motion is profitable. The heuristics are safe; they do not generate adverse affects; however, certain optimizations may occur in Schwartz which Schaefer would deem inadvisable.

We now describe the global optimization algorithm drawing freely from both Schaefer and Schwartz. During the semantic phase of the compiler, the program flow graph for the routine being compiled is generated. During the local optimization

phase, big sectors for the most frequently occurring expressions will be constructed; these big sectors will express which variables have been killed, which have been computed in a block after all uses, and which variables are not used in a block at all. Also, during the local optimization phase the vectors for computation of live variables and available computation can be initialized. At the beginning of the global optimization phase, the program flow graph is divided into intervals, the resulting reduced graph is again divided into intervals, and so forth until the flow graph has been completely reduced. The result will either be one node in most cases or an irreducible flow graph. The first set of equations to be solved will be the redundant expression or available expression equations. These equations express what expressions are available on exit from each of the basic blocks in the program flow graph. This algorithm is described in detail in Schwartz. It involves processing each interval in order, starting with the first interval in the program and continuing, and involves two passes. The first computes within an interval what expressions are available on entry to the interval. The same process is then applied to the reduced graph and so forth until the totally reduced graph is reached. If this totally reduced graph is one node, the expressions available on entry to that block, namely the whole

routine, can be used and propagated back down through the whole set of reduced graphs to obtain a solution for the available computation equation. If the totally reduced graph is not one node, the iterative equation method will be applied to it as described in Allen. This will then give us a solution for a totally reduced graph, and that again can be propagated back through a sequence of reduced graphs until a set of solutions for the available computation equations is obtained.

At this point the redundant computations can be eliminated. Each such computation is marked unnecessary in the internal form of the program. Besides this elimination those computations which are unreachable will be reported to the error routine as warning messages to the programmer.

Now the code motion and strength reduction optimizations are applied. First, the program must be divided into loops. This is done by taking as a loop the strongly connected component of any interval and then in the reduced graph the strongly connected component of any interval there, and so forth, providing us with a nested set of constructs similar to the program loops. Using the available computation information from the first part of the global optimization phase, those expressions that are movable out of the loop are moved to the head of the loop; in particular, the

head of the interval. This optimization is discussed in Schwartz [Schwartz]. Only those optimizations which are safe and possible are moved. In particular, an expression is safe to move only if it is evaluated no matter what path is taken through the loop. The order of motion is as follows. First, the innermost loops are analyzed and expressions are moved out of the innermost loop. Then strength reduction is applied to the innermost loop; then linear test analysis is applied to that loop. After all this has been done, the head of the loop contains new information. The local optimization of the block which is the head of the loop is reexecuted to make sure that common subexpressions are eliminated locally from that head. This process is applied to each lowest level loop and then to those loops that contain the lowest level of loops on up to the whole routine. If the whole routine is irreducible, the whole routine itself is handled as one large loop and code can be moved out of it but not out of any subintervals in it in the irreducible part.

Strength reduction consists of replacing multiplications by additions when this is profitable. This is particularly useful in subscript evaluation and other occurrences of expressions of the form integer times a constant or integer plus a constant. There is a full discussion of this technique in Schwartz. Briefly, it consists of replacing occurrences of integer times a

constant by a new temporary variable which is kept updated to the value of this integer times a constant. When the integer is incremented, then the temporary variable is incremented by the multiplier; when it is decremented, it is decremented by the multiplier. All occurrences of the integer times that constant are replaced by this temporary variable. Frequently, all uses of the original variable will be removed and replaced by uses of this temporary variable. In that case, the variable itself can be removed. The net result is that a variable involved in a multiplication has been replaced by a variable involved in an addition. Linear test replacement is the extension of this idea to the conditional branches at the end of the loop. Frequently, they will be of the form integer greater than a constant, or integer equal to a constant. If the integer is replaced by a variable which is a positive multiple of that integer, the opposite side of the relation can be replaced by the same integer multiple and the whole relation can be replaced by one involving the temporary variable. After strength reduction, frequently the only occurrences of the variable are in the relational expressions determining the end of the loop. Thus, should linear test replacement remove these occurrences too, the variable may be removed completely.

After these optimizations have been performed, certain variables will have been assigned and no longer used. This can ccur without program error. For instance, the local optimization phase replaces uses of variables by uses of variables with the same value. It may occur that the assignment o the variable is no longer used at all; therefore that assignment can be eliminated. If all assignments to a variable an be eliminated, the variable can be deleted completely from the program. To do this, the liveness equations for each variable must be solved. This is a set of equations discussed in [Schaefer] and in [Cocke-70]. It consists of computing for each variable and for each basic block in the program whether at the end of that basic block there exists a path to a use of the variable which does not include an assignment to the variable. If such a path exists, the variable is declared to be alive; if no such path exists, the variable is dead at that point. If the variable is dead after an assignment, then the assignment is irrelevant. The solution of the liveness equations is similar to the solution of the variable computation equations. This is discussed in Schaefer and Schwartz; however, the program is traversed in reverse rather than forward. Once these equations have been solved, a linear scan of the program will eliminate all assignments to variables that do not have uses. These

assignments are traversed in reverse order and the assignment is marked unnecessary if it is dead. This terminates the implemented optimizations in the global optimization phase.

Other optimizations could have been performed in the global optimization phase. One is loop fusion. This consists of taking two similar loops and combining them into one loop. This can be done if the number of iterations are the same for both loops and no instructions occur between these two loops that could adversely affect the union of the two loops.

Another optimization is splitting. This involves replicating the body of the loop two or more times and decreasing the number of times through the loop accordingly. This optimization can be performed on loops that have a fixed number of times executed. These two optimizations together provide a fairly powerful way of optimizing a program; however, no known algorithm efficiently implements them. The information for the implementation of such algorithms is available, provided by the semantic phase. Some algorithms do exist; some of them are discussed in Wagner [Wagner]. However, at this time we do not envision using these optimizations except in the simple cases where the loops that are fused are those involved in adding two arrays, subtracting two arrays, or similar operations on arrays.

[Allen-76]
F. E. Allen, "Annotated Bibliography of Selected Papers on Program Optimization", IBM Research Report RC5889, T. J. Watson Research Center, Yorktown Heights, N.Y., March 1976

[Allen-76a]
F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", CACM Vol. 19 No. 3, pp. 137-147, March 1976

[Allen-74]
F. E. Allen, Ken Kennedy, "Reduction of Operator Strength", Rice Technical Report 476-093-6, Rice University, August 1974

[Allen-72]
F. E. Allen and J. Cocke, "Graph-Theoretic Constructs for Program Control Flow Analysis", IBM Research Report RC3923, T. J. Watson Research Center, Yorktown Heights, N.Y., July 1972

[Earnest]
C. Earnest, "Some Topics in Code Optimization", J. ACM 21(1) Jan 74, pp 76-102

[Kennedy-71]
K. Kennedy, "an Algorithm for Common Subexpression Elimination and Code Motion", SETL Newsletter #28, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., May 1971

[Kennedy-71a] K. Kennedy, P. Owens, "An Algorithm for Use-Definition Chaining", SETL Newsletter #37, , Courant Institute of Mathematical Sciences, New York University, New York, N.Y., July 1971

[Kennedy-72]
K. Kennedy, "Algorithm for Live-Dead Analysis Including Node-Splitting", SETL Newsletter #38, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., January 1972

[Kennedy-73]
K. Kennedy, "Reduction in Strength Using Hashed Temporaries", SETL Newsletter #102, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., March 1973

[Kennedy-73a]
K. Kennedy, "Linear Function Test Replacement", SETL Newsletter #107, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., May 1973

[Kennedy-73b]
K. Kennedy, "Global Dead Computation Elimination", SETL Newsletter #111, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., August 1973

[Kennedy-73c]
K. Kennedy, "An Algorithm to Compute Compacted Use-Definition Chains", SETL Newsletter #112, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., August 1973

[Kennedy-74]
K. Kennedy, "Variable Subsumption with Constant Folding", SETL Newsletter #123, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., February 1974

[Kennedy-74a]
K. Kennedy, "Schaefer's Node Splitting Algorithm", SETL Newsletter #125, Courant Institute of Mathematical Sciences, New York University, New York, N.Y., February 1974

[Kennedy-76]
K. Kennedy, "A Comparison of Two Algorithms for Global Data Flow Analysis", SIAM Journal of Computing, Vol. 5, No. 1, pp 158-160, March 1976.

[Kildall]
G. A. Kildall, "lobal Expression Optimization During Compilation", Proc. ACM Conf. on Principles of Programming Languages, Oct 73, pp 194-206

[Cocke-69]
J. Cocke and Raymond Miller, "Some Analysis Techniques for Optimizing Computer Programs", Proc. Second Intl. Conf. Of Systems Sciences, Hawaii, Jan. 1969

[Cocke-70]

John Cocke and J. T. Schwartz, "Programming Language and their Compilers", Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, N.Y., April 1970

[Schwartz]
Jacob T. Schwartz, "On Programming: An Interim Report on the SETL Project, Installment II: The SETL Language and Examples of Its Use", Courant Institute of Mathematical Sciences, New York University, New York, N.Y., 1973

[Schaefer]
M. Schaefer, "A Mathematical Theory of Global Program Optimization", Prentice-Hall, Englewood Cliffs, New Jersey, 1973

[Wagner]
Robert Alan Wagner, "Some Techniques for Algorithm Optimization with Application to Matrix Arithmetic Expressions", PhD. Thesis Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1968

[Tarjan]
Robert E. Tarjan, "Solving Path Problems on Directed Graphs", Stanford Computer Science Technical Report STAN-CS-75-528, Stanford University, Stanford California, October 1975

[Graham]
S. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis", Conference Record of Second ACM Symposium on Principles of Programming Language, pp. 22-34, 1975

[Ullman]
J. D. Ullman, "A Fast Algorithm for the Elimination of Common Subexpressions", Acta Informatica, Vol. 2, pp. 191-213, 1973

Chapter 9

Register Allocation and Code Generation Phases

The register allocation and code generation phases are combined into one pass. The code generation phase is a co-routine with the register allocation phase. Actually, the register allocation phase consists of multiple passes, the first being used to determine look-ahead information on variables, the second to estimate what variables are in registers, the third to generate code and assign variables to symbolic registers, and finally to assign physical registers to the symbolic registers. We will discuss these two phases independently. However, we must remember that code generation may be considered a part of this pass.

Four register allocation schemes for the compiler have been evaluated for possible use in the COL, those described in [Day], [Beatty], [Freiberghouse] and [Harrison]. Day proposes an algorithm for assigning n variables to m registers where n is greater than m. The algorithm assigns these variables based on the value of having the variable in a register. The paper presents both an optimal but slow algorithm and a non-optimal but fast and simple algorithm for this assignment. He argues statistically that the fast algorithm is 90% as good as the

optimal one. As was pointed out by Beatty [Beatty], Day's algorithm has the problem of unifying global register allocation and local register allocation. If local register allocation is done before global register allocation, it is difficult to hold values in registers over a long period of time. If the opposite is done, inferior code is produced.

Beatty presents a fairly complex algorithm which is global, like Day's. It appears to be more practical, however, in several ways. It appears to fit in well with other optimization activities in a compiler. The thrust of the algorithm is to compile simple-minded code with a simple local register allocation scheme, then optimize the results by moving the load and store instructions around in a way similar to other code motion optimization activities.

As the Freiberghouse algorithm is used for local allocation only with no flow control, it is considerably less general. It is not shown to be optimal but is supported statistically.

Harrison's algorithm is a generalization of a storage allocation algorithm by Belady [Belady]. He determines an approximation to the access structure to the variables in the program. Then using this access structure he approximates the variables that must be in registers at each point. He then

generates code making this assumption and following that assigns particular registers to each symbolic register previously determined. Like Beatty's algorithm, this algorithm has a combinatorial nature to it. However, the combinatorial nature can be minimized by using various structures within the compiler. It seems to be comparable to Beatty's and the space taken seems to be comparable if not smaller.

The choice of register allocation for the COL is determined by the requirements of communications and systems programming. On multi-register computers, the code generated for register loading and removal can be as large as that for the computations involved. Thus, if a good register allocation algorithm is not used, the code generated will suffer seriously. On the other hand, the optimizations that may be performed on code written for communications or systems programming frequently do not have many redundant computations or computations to be moved. These can only be done for arrays, accesses, and for some pointer accesses. All this implies that register allocation is as important as the other global optimizations, and hence the time required to perform good global register allocation is spent profitably. With this criterion, we investigate the four algorithms. Of the four, Beatty's algorithm and Harrison's algorithm seem the best to adopt in an optimizing compiler. The algorithm should fit in

well with other code motion activities. The computations required, though not trivial, do not appear to be unreasonable. Two advantages over Day's method are apparent. The Day algorithm insists on uniform assignment through the region; a variable is always in the same register. Second, the Day algorithm does not rely on partial local allocation. Such allocation is usually required, because in most machines one is not free to use either memory or registers interchangeably in many instructions. In a complex example given by Beatty, the Day algorithm cannot achieve as good results using his optimal algorithm. The Freiberghouse algorithm is probably a good candidate for the local allocation phase, preliminary to the Beatty or Harrison global allocation. However, the type of machine will have to be considered here too. The running time for the four algorithms is probably acceptable, given that we consider register allocation as important as other global optimizations. Thus, of the four algorithms we come to a choice between Beatty's and Harrison's. From independent results [Yhap], these two algorithms seem comparable in the quality of code generated. Harrison's algorithm probably takes less space to implement. Beatty reports execution times of half a second per reference using PL/1 on a 360 Model 67. That may be too high based on some preliminary programming of his algorithm done at BBN. More than that, some of the computation Beatty requires

for full analysis, variable dependencies, etc., will almost certainly be required anyway in an optimizing compiler. We expect Harrison's results to be similar.

We have chosen Harrison's algorithm for the following reasons. First, the storage space needed seems to be smaller; second the algorithm seems to fit into the classic code generation method more easily than Beatty's; and third, if the algorithm does turn out to be too expensive to use in an operational compiler, there are schemes which may be used to replace the algorithm by an approximate algorithm with better computation time and similar results. Furthermore, Harrison's algorithm seems to fit into the standard compiler framework better. In particular, it need not be known ahead of time whether a variable will be in a register or in memory to perform Harrison's algorithm.

[Day]
W. H. Day, "Compiler Assignment of Data Items to Registers", IBM Systems Journal, Vol. 9, No. 4, pp. 281-317, 1970

[Beatty]
J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code", IBM Journal of Research and Development, Vol. 18, No. 1, pp. 20-39, January 1974

[Freiberghouse]
R. A. Freiberghouse, "Register Allocation Via Usage Counts", CACM, Vol. 17, No. 11, pp. 638-642, November 1974

[Harrison]
W. Harrison, "A Class of Register Allocation Algorithms", IBM Research Report RC5342, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, March 1975

[Yhap]
E. F. Yhap, "General Register Assignment in Presence of Data Flow", IBM Research Report RC5645, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, September 1975

[Belady]
L. A. Belady, "A Study of Replacement Algorithms for Virtual-Storage Computer", IBM Systems Journal, Vol. 15, No. 2, 1966

Chapter 10

Code Generation Phase

The code generator will be called as a co-routine for the register allocation system. It will be called in several modes. The first involves no generation of code but determines what variables will be needed in registers and what variables can efficiently reside in memory; the second mode consists of the actual code generation; and the third mode consists of fixing the symbolic registers used in code generation to physical real registers.

We have surveyed several methods of code generation, including the FORTRAN Level H bit masking method, hand coded code generators, interpreted code generation languages, and finite state machines incorporating an interpretive code generation language.

The COL compiler must be capable of generating code using a fixed number of operators but a large number of data sizes and addressing modes. Typically, hand coded generators for this form of language are large and bulky. Furthermore, such code generators have little machine independence (if any code generation phase can have machine independence). The FORTRAN code generation method is good for a few data types and accessing

methods; however, in the general case it must be bulky and slow. Similarly, the interpretive code generation language becomes bulky, though less so. Wilcox [Wilcox] has suggested a compromise involving finite state machines with an interpreter. We propose the implementation of his method as the code generation phase of the COL compiler with two modifications. The first modification is necessary to allow different modes of execution, as just described. The other modification involves the fact that register allocation will be done external to the code generation phase in the register allocation phase (except for local register allocation). Thus, the complicated attribute handling system involved in Wilcox's multipass optimization scheme can be simplified so that attributes are determined externally and the one pass scheme is applied.

The basic structure of the code generation phase is as follows. When the register allocation phase walks through the program, it will call the code generation phase at each statement in the internal form. Recall that each statement in the internal form is either a procedure call, a branching statement, or an assignment statement of some form. In other words, it is a simple statement with a tree representing each expression. The code generation phase consists of a small program which walks through the expression trees of the statement and generates code.

At each node in execution order walk, it will call an appropriate routine of the interpretive language. The interpretive language consists of a set of statements designed to generate code; there will be a statement to generate each class of instructions, there will be a statement to arrange that certain addresses are addressable, there are statements to insure that certain items are in registers. The languages will be full and general in the sense of code generation. The interpreter will be able to check whether certain variables are in registers of a particular class (e.g., is this variable in an even or odd register?) and it will be able to generate move instructions to move variables around and keep track of where they are. The code generator will generate another internal form for the program. This internal form will be a direct representation of the machine language to be generated for the machine. It will probably include the opcode, the registers, and the addresses, and a one-to-one correspondence to their occurrence in the object machine we are generating code for. However, there may not be the exact binary generated. In particular, variables will be represented by pointers to the symbols table, registers will either be represented as symbolic registers to be later assigned or as the actual physical registers involved, and the opcodes may not be the actual binary opcode that will be generated but may be an

index into a table of routines or into a table of symbols. All labels that are generated by the code generation phase will be included in the symbol table and will involve pointers both to the location of the statement they label and to all uses of this label. This will be invaluable later when the peephole optimization phase will attempt to reduce the code that has been generated.

By personal experience, we know code generation to be a complex process. The complexity derives from the enormous number of special cases. Guaranteeing that the correct code is generated in each case is difficult. Fortunately, many code generation problems can be reduced to decision tables. In the case of the COL, each expression node and statement node requires a code-generation language routine. These routines can be stated in decision table form. We propose that a decision table compiler be used to maintain the code generators. Such techniques are discussed in the first three of the following references:

[Schumacher]
Helmut Schumacher, "The Synthesis of Optimal Decision Trees from Decision Tables", University of Toronto Computer Systems Research Group Techn. Rep. CSRG-46, December 1974

[Myers]
H.J. Myers, "Compiling Optimized Code from Decision Tables", IBM Journ. Res. and Developm., pp.489-503, Sept. 1972

[Baker]
Brenda S. Baker, "An Algorithm for Structuring Flow Graphs", JACM Vol. 24, No. 1, pp. 98-120, January 1977

[Wilcox]
Thomas Richard Wilcox, "Generating Machine Code for High-Level Programming Languages", Ph.D Thesis, Cornell University, Ithaca, New York, September 1971

Chapter 11

Peephole Optimization

Even with sophisticated code generation, significant improvements in generated code can be made with a scan of the generated code replacing sequences of instructions by shorter sequences with the same effect. This approach was proposed by McKeeman[Mckeeman] and extended by BLISS-11[Wulf].

The FINAL phase of the BLISS-11 compiler is an excellent model for this phase of the COL compiler. That phase makes multiple scans of the code generated by the code generation module. It performs such modifications as:

- replace sequence by shorter equivalent sequences
- eliminate unreachable code
- merge identical code sequences (cross jumping)
- replace addressing modes by faster equivalent ones

This scan is repeated until no changes in the code occur.

Experience with BLISS-11 convinces us that one additional optimization can be performed which will reduce code by a significant factor. BLISS-11 performs the FINAL phase on each routine separately. The beginning of each routine involves saving the registers used and the end of each routine involves the restoration of registers used. Frequently the first or last action by a routine is a routine call. By performing peephole

optimization on all routines simultaneously, these register saves and restores can be optimized also. From personal experience, an additional 5% saving of code spacing will be gained by this action.

The implementation of this phase of the compiler is a straightforward repetitive scan of the object code. However, some machines have a very complex instruction set or collection of addressing modes. To speed the implementation and execution of this phase, the code sequence can be viewed as a string (ignoring the accumulators and addresses). The optimizations can be looked upon as substrings to be searched for. Then the algorithm of Aho[Aho] for searching for all occurrences of a set of substrings in a large string can be used as a driver for the peephole optimization search. This approach guarantees that the specified sequences will be spotted and provides an easy mechanism for adding and subtracting optimizations from the collection of optimizations used. To decrease the size of the finite state machine generated, the instruction set should be partitioned into equivalence classes of instruction codes, the equivalence being defined by the property of occurring in similar peephole optimizations.

[McKeeman]
W. M. McKeeman, "Peephole Optimization", CACM Vol. 8, pp. 443-444, July 1965

[Wulf]
William A. Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke, "The Design of an Optimizing Compiler", American Elsevier, New York, 1975

[Aho]
Alfred V. Aho and Margaret J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search", CACM Vol. 18, No. 6, pp. 333-340, June 1975

Chapter 12

Assembly Pass

After the code generation phase of the compiler is complete, the assembly pass must be performed. Its function is to determine the addresses of each datum and label, to replace abstract representations of each of these by their actual values, and to generate the output module for the loader. The latter is dependent upon both the machine characteristics of the object machine and the source machine. The dependence on the object machine is obvious; the length of instructions and their form cannot be machine independent. The same applies to the addressing characteristics of branches and data. Equally machine dependent is the format of the object module produced. Its form depends on the operating system of the host machine, the characteristics of the object machine, and the form of the loader. However, some general guidelines can still be specified and a model for this phase can still be written.

The COL compiler must directly generate the object module. It is not acceptable for the COL compiler to generate a file to be used as input to an assembler. It must be able to generate an assembly listing of the code; however, this listing must not be capable of assembly or all security and debugging features of the COL will have been nullified.

Chapter 13

Pluribus and DECSystem-20 Examples

Although the COL language has been designed to ease the programming and maintenance tasks for communications systems, it will not be used if the run-time costs in space and time are too expensive. To guarantee that these costs are not excessive we have modelled the compilation of sample COL programs. The analysis for the IBM 360 is given in chapter 14. This chapter analyzes the object files produced by the proposed COL compiler for both the DECsystem-20 and BBN Pluribus computers. First we will discuss the basic structure of these two machines. Then we will present a proposed run-time structure for each machine. Finally we will present the object modules produced for both machines from the compilation of the example QUICKSORT given in Chapter 1.

The DECsystem-20 computer is a descendant of the PDP-10 computer. It is a 36 bit per word computer with 256K words of memory. Each word of memory is directly addressable without base registers. There are 16 fast registers which can be used as accumulators. Fifteen of these can be used as index registers. There is a rich collection of machine instructions including stack handling instructions and general byte instructions. This

AD-A047 393 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS F/G 9/2
COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE IMPLEMENTATION--ETC(U)
MAY 77 C R MORGAN, A EVANS DCA100-76-C-0051
UNCLASSIFIED BBN-3533 SBIE-AD-E100006 NL

2 OF 2
AD
A047393

END
DATE
FILMED
1 -78
DDC

computer is designed for easy, efficient programming; in fact generating a compiler is not difficult. However, it is very difficult to build an efficient compiler, since the rich collection of instructions and the general addressing structure make it difficult for a compiler to do as well as a clever assembly language programmer. Our example shows that the expense is not too great and the readability of the code is much higher for a COL program.

The BBN Pluribus computer is an asynchronous multiprocessor built from Lockheed SUE processors. This is a 16 bit per word computer with 512K possible words. The memory can be addressed as 16-bit words or 8-bit bytes. There are 7 fast registers which can be used as accumulators or index registers. The processors have a simple instruction set with a fairly general addressing structure. The processor can push an accumulator onto a stack or pop the top of a stack into an accumulator. Each processor has a 32K word virtual address space. This space is divided into local memory and shared memory. Each shared memory area is controlled by a map register which indicates the location of the shared memory page in the large shared memory.

The COL run-time structure on the DECsystem-20 is similar to the typical assembly language structure on that machine. In

fact, it is the BLISS-11 run-time structure modified for the DECsystem-20 instead of the PDP-11. There is a stack controlled by one of the 16 fast registers. All dynamic variables either reside in registers or on the stack. If a dynamic variable is on the stack it is addressed by its position relative to the end of the stack at that point in the program. On encountering a declaration in a program the stack pointer is adjusted to include the dynamic variables declared in the declaration. The static variables are stored in fixed locations in memory. They can be initialized when the modules are linked and can be referenced by their address in memory at any location within the scope of their declaration. On routine or function entry the first few arguments will be passed to the routine or function in registers. Large arguments and the excess arguments will be passed on the stack. For a function call space is allocated on the stack or in a register for the return value before the function is called. These stack manipulations match the scoping rules given in the COL. The free storage area builds from the opposite end of memory towards the stack. A library routine will implement the storage management mechanism for the program. It will keep track of free areas and allocate to the program memory areas of arbitrary size. There is no garbage collection as such. The applications program is responsible to return free storage when not in use or provide a garbage collection mechanism.

The run-time system for the Pluribus is similar. The only added detail is the manipulation of map registers for shared memory. The Pluribus programmer handles this problem by assigning one map register for each of code, variables, and free storage. Although we doubt that this is the optimum method of utilizing map registers, we will adopt the same convention for the COL compiler.

We will now analyze the QUICKSORT example for the DECsystem-20 computer. The source program is read by the lexical analyzer and broken into atoms. These atoms are fed to the syntax analyzer which calls the semantics analyzer. The semantics analyzer generates the internal form of the program. This internal form is broken into straight-line sequences of code called basic blocks. Symbolic representations of these forms are presented in the tables that follow. Next the local optimization phase is called. In this particular program the only optimizations performed are the replacement of subscripts by the actual offsets required by the object machine and the replacement of self modifications such as I:=I+1 by I*=+1. The first of these optimizations later leads to global optimizations by strength reduction. The second optimization leads to better code generation. Next the global optimization phase is called. The various derived graphs used by the interval analysis algorithms

are computed. There are no benefits from global constant propagation and common subexpression elimination in this example. There are great benefits from strength reduction. Strength reduction replaces each index by the equivalent pointer. These pointers are incremented and decremented where the corresponding index was incremented or decremented. The pointer to A[I] is named AI2, and the other pointers are similarly named. Next the register allocation phase is called. It allocates storage for each variable and assigns registers to expressions at each point in the program. In this example and computer there are enough registers to hold all small variables. The array STACK is stored as a dynamic variable on the stack. Finally code is generated as presented in the final table for the DECsystem-20 example. Following code generation peephole optimization is performed with little benefit in this example. However, if the variable STACK or A were stored as parallel arrays then the code sequences:

```
ADDI   AI2,2
JRST   L$3
```

could be replaced by the single instruction AOJA L$3.

Internal Text Form of QUICK.COL

```
  [1]       start_routine [43]
  [2] L$0:  S:=1
  [3]       STACK[1].L:=1
  [4]       STACK[1].R:=N
  [5]       start_repeat [42]
  [6] L$1:  L:=STACK[S].L
  [7]       R:=STACK[S].R
  [8]       S:=S-1
  [9]       start_repeat [40]
 [10] L$2:  I:=L
 [11]       J:=R
 [12]       KEY:=A[(L+R)/2].KEY
 [13]       start_repeat [31]
 [14]       start_while [18]
 [15] L$3:  if A[I].KEY >= KEY goto L$4
 [16]       I:=I+1
 [17]       goto L$3
 [18]       end_while [14]
 [19]       start_while [23]
 [20] L$4:  if KEY >= A[J].KEY goto L$5
 [21]       J:=J-1
 [22]       goto L$4
 [23]       end_while [19]
 [24]       start_if [29]
 [25] L$5:  if I > J goto L$6
 [26]       swap(A[I],A[J])
 [27]       I:=I+1
 [28]       J:=J-1
 [29]       end_if [24]
 [30] L$6:  if I <= J goto L$3
 [31]       end_repeat [13]
 [32]       start_if [37]
 [33]       if I >= R goto L$7
 [34]       S:=S+1
 [35]       STACK[S].L:=I
 [36]       STACK[S].R:=R
 [37]       end_if [32]
 [38] L$7:  R:=J
 [39]       if L < R goto L$2
 [40]       end_repeat [9]
 [41]       if S <> 0 goto L$1
 [42]       end_repeat [5]
 [43]       end_routine [1]
```

Basic Blocks for QUICK.COL

```
Block 1:    [2] L$0: S:=1
            [3]      STACK[1].L:=1
            [4]      STACK[1].R:=N

Block 2:    [6] L$1: L:=STACK[S].L
            [7]      R:=STACK[S].R
            [8]      S:=S-1

Block 3:   [10] L$2: I:=L
           [11]      J:=R
           [12]      KEY:=A[(L+R)/2].KEY

Block 4:   [15] L$3: if A[I].KEY >= KEY goto L$4

Block 5:   [16]      I:=I+1
           [17]      goto L$3

Block 6:   [20] L$4: if KEY >= A[J].KEY goto L$5

Block 7:   [21]      J:=J-1
           [22]      goto L$4

Block 8:   [25] L$5: if I > J goto L$6

Block 9:   [26]      swap(A[I],A[J])
           [27]      I:=I+1
           [28]      J:=J-1

Block 10:  [30] L$6: if I <= J goto L$3

Block 11:  [33]      if I >= R goto L$7

Block 12:  [34]      S:=S+1
           [35]      STACK[S].L:=I
           [36]      STACK[S].R:=R

Block 13:  [38] L$7: R:=J
           [39]      if L < R goto L$2

Block 14:  [41]      if S <> 0 goto L$1

Block 15:  [43]      end_routine [1]
```

Basic Blocks for QUICK.COL after Local Optimization

```
Block 1:    [2] L$0: S:=1
            [3]      STACK[2].L:=1
            [4]      STACK[2].R:=N

Block 2:    [6] L$1: L:=STACK[2*S].L
            [7]      R:=STACK[2*S].R
            [8]      S*=-1

Block 3:   [10] L$2: I:=L
           [11]      J:=R
           [12]      KEY:=A[(L+R)/2].KEY

Block 4:   [15] L$3: if A[2*I].KEY >= KEY goto L$4

Block 5:   [16]      I*=+1
           [17]      goto L$3

Block 6:   [20] L$4: if KEY >= A[2*J].KEY goto L$5

Block 7:   [21]      J*=-1
           [22]      goto L$4

Block 8:   [25] L$5: if I > J goto L$6

Block 9:   [26]      swap(A[2*I],A[2*J])
           [27]      I*=+1
           [28]      J*=-1

Block 10:  [30] L$6: if I <= J goto L$3

Block 11:  [33]      if I >= R goto L$7

Block 12:  [34]      S*=+1
           [35]      STACK[2*S].L:=I
           [36]      STACK[2*S].R:=R

Block 13:  [38] L$7: R:=J
           [39]      if L < R goto L$2

Block 14:  [41]      if S <> 0 goto L$1

Block 15:  [43]      end_routine [1]
```

Basic Blocks for QUICK.COL after Global Optimization

```
Block 1:    [2] L$0: STACKP:=REF(STACK[2])
            [3]      STACK[2].L:=REF(A[2])
            [4]      STACK[2].R:=REF(A[2*N])

Block 2:    [6] L$1: AL2:=STACKP@.L
            [7]      AR2=STACKP@.R
            [8]      STACKP*=-2

Block 3:   [10] L$2: AI2:=AL2
           [11]      AJ2:=AR2
           [12]      KEY:=((AL2+AR2)/2)@.KEY

Block 4:   [15] L$3: if AI2@.KEY >= KEY goto L$4

Block 5:   [16]      AI2*=+2
           [17]      goto L$3

Block 6:   [20] L$4: if KEY >= AJ2@.KEY goto L$5

Block 7:   [21]      AJ2*=-2
           [22]      goto L$4

Block 8:   [25] L$5: if AI2 > AJ2 goto L$6

Block 9:   [26]      swap(AI2@,AJ2@)
           [27]      AI2*=+2
           [28]      AJ2*=-2

Block 10:  [30] L$6: if AI2 <= AJ2 goto L$3

Block 11:  [33]      if AI2 >= AR2 goto L$7

Block 12:  [34]      STACKP*=+2
           [35]      STACKP@.L:=AI2
           [36]      STACKP@.R:=AR2

Block 13:  [38] L$7: AR2:=AJ2
           [39]      if AL2 < AR2 goto L$2

Block 14:  [41]      if STACKP <> REF(STACK[0]) goto L$1

Block 15:  [43]      end_routine [1]
```

POSSIBLE CODE GENERATED BY CODE GENERATION PHASE

```
A=1
N=2
STACKP, AI2, AJ2,AR2, AL2, KEY, TMP in registers 3-9

QUICK:  Save Registers 3-9
        MOVEI   P,24(P)          ; ALLOCATE SPACE ON THE STACK
        MOVEI   STACKP,-23(P)    ; [2]
        MOVE    TMP,2(A)         ; [3]
        MOVEM   TMP,-23(P)
        MOVE    TMP,N            ; [4]
        ADD     TMP,N
        ADD     TMP,A
        MOVEM   TMP,-23(P)

L$1:    MOVE    AL2,(STACKP)     ; [6]
        MOVE    AR2,1(STACKP)    ; [7]
        SUBI    STACKP,2         ; [8]
L$2:    MOVE    AI2,AL2          ; [10]
        MOVE    AJ2,AR2          ; [11]
        MOVE    TMP,AL2          ; [12]
        ADD     TMP,AR2
        ASH     TMP,-1
        MOVE    KEY,(TMP)
L$3:    CAMG    KEY,(AI2)        ; [15]
        JRST    L$4
        ADDI    AI2,2            ; [16]
        JRST    L$3              ; [17]
L$4:    CAML    KEY,(AJ2)        ; [20]
        JRST    L$5
        SUBI    AJ2,2            ; [21]
        JRST    L$4              ; [22]
L$5:    CAMLE   AI2,AJ2          ; [25]
        JRST    L$6
        EXCH    TMP,(AI2)        ; [26]
        EXCH    TMP,(AJ2)
        EXCH    TMP,(AI2)
        ADDI    AI2,2            ; [27]
        SUBI    AJ2,2            ; [28]
L$6:    CAMG    AI2,AJ2          ; [30]
        JRST    L$3
        CAML    AI2,AR2          ; [33]
        JRST    L$7
        ADDI    STACKP,2         ; [34]
```

```
        MOVEM   AI2,(STACKP);   ; [35]
        MOVEM   AR2,1(STACKP)   ; [36]
L$7:    MOVE    AR2,AJ2         ; [38]
        CAMGE   AL2,AR2         ; [39]
        JRST    L$2
        CAIE    STACKP,-23(P)   ; [41]
        JRST    L$1
        MOVEI   P,-24(P)        ; CLEAR THE STACK
        Restore registers 9-3
        POPJ    P,              ; RETURN FROM SUBROUTINE
```

The compilation for the Pluribus is similar to that for the DECsystem-20, with two exceptions. The arrays are indexed by byte address so all the multiples of 2 are replaced of multiples by 4. To indicate this the variable AI2 is replaced by AI4. The other difference is in code generation. A processor for a Pluribus does not have enough registers to hold all variables in fast memory. The variables which are candidates for remaining in registers are:

L-Value of A	N	STACKP	AI4
AJ4	AR4	AL4	
KEY	TMP		

The L-value of A and the value of N are needed only at the beginning of the program. They are assigned to registers 2 and 3 respectively. Variables AL4, AR4, AJ4, and AI4 are assigned to registers 2 through 5. Register 6 is used for a temporary variable and register 7 is used for STACKP. Register 1 is used for the stack pointer for dynamic variables. There are two

variables on the stack, STACK and KEY. The stack on the Pluribus expands in memory toward location zero. This is done so that the stack pointer is always pointing to the latest entry in the stack. KEY is at the top of the stack. With these modifications the compilation process is the same as the compilation process for the DECsystem-20. The following code is a possible result of the compiler.

POSSIBLE CODE GENERATED FOR THE PLURIBUS

```
QUICK:  SAVE REGISTERS 2-7
        SUB     %1,=50          ; ALLOCATE SPACE ON THE STACK
        LDA     STACKP,=2(%1)   ; [2]
        LDA     TMP,=4(A)       , [3]
        STA     TMP,=2(%1)
        SLA     N,2             ; [4] - N NO LONGER USED
        ADD     N,A             ; A NO LONGER USED
        STA     N,=4(%1)
L$1:    LDA     AL4,(STACKP)    ; [6]
        LDA     AR4,2(STACKP)   ; [7]
        SUB     STACKP,=4       ; [8]
L$2:    LDA     AI4,AL2         ; [10]
        LDA     AJ4,AR4         ; [11]
        LDA     TMP,AL4         ; [12]
        ADD     TMP,AR4
        SRA     TMP,1
        LDA     TMP,(TMP)
        STA     TMP,(%1)
L$3:    LDA     TMP,(AI4)       ; [15]
        CMP     TMP,(%1)
        BG      L$4
        ADD     AI4,=4          ; [16]
        BR      L$3             ; [17]
L$4:    LDA     TMP,(AJ4)       ; [20]
        CMP     TMP,(%1)
        BL      L$5
        SUB     AJ4,=4          ; [21]
        BR      L$4             ; [22]
L$5:    CMP     AI4,AJ4         ; [25]
        BG      L$6
        STA     STACKP,-(%1)    ; [26]
        LDA     STACKP,(AI4)
        LDA     TMP,(AJ4)
        STA     TMP,(AI4)
        STA     STACKP,(AJ49
        LDA     STACKP,2(AI4)
        LDA     TMP,2(AJ4)
        STA     TMP,2(AI4)
        LDA     STACKP,(%1)+
        ADD     AI4,=4
        SUB     AJ4,=4
L$6:    CMP     AI4,AJ4         ; [30]
        BLE     L$3
```

```
        CMP     AI4,AR4             ; [33]
        BGE     L$7
        ADD     STACKP,=4           ; [34]
        STA     AI4,(STACKP)        ; [35]
        STA     AR4,2(STACKP)       ; [36]
L$7:    LDA     AR4,AJ4             ; [38]
        CMP     AL4,AR4             ; [39]
        BL      L$2
        CMP     STACKP,%1           ; [41]
        BNE     L$1
        ADD     %1,=50              ; CLEAR THE STACK
        RESTORE REGISTERS 2-7
        JMP     (%7)                ; RETURN FROM SUBROUTINE
```

Chapter 14

How COL compiles for the IBM 360

In this chapter we examine the implementation problems, and solutions thereof, for a hypothetical compiler for the IBM-360 (and IBM-370) version of COL. The basic model of the COL compiler is designed to be the same across all machines, using the most up-to-date techniques in compiling and optimization. Thus we concentrate on the particulars of using the machine, including data representation, code generation and the run time system.

We approach this discussion by presenting two sample programs and their code for the 360, obtained by modeling the COL compiler. Relating the discussion to the sample programs provides examples that are consistently developed. The sample programs are reasonably comprehensive. In some form, they overview the code generated for all of COL's syntax structures as well as necessary run time organization and data representation. Additional examples for particular cases are provided, but the three main examples provide the reader with a consistent reference.

Following the three COL examples and their code we examine some basic properties of the IBM-360. Data representation for COL

is then addressed, followed by a development of the runtime system, and a discussion of code generation and optimization.

14.1 Sample programs

14.1.1 Sample Program PRINT

The first sample program can be found in section 7.2 of the COL Manual. The line numbers to the left of the COL statements and to the far right of the 360 code are used only for reference in the text and are not parts of the programs. A one or two digit reference, such as line 12, refers to the COL statements and a three digit reference refers to the code, e.g., line 071. When citing different samples at the same time this number is preceded by the first letter of the sample, e.g., line P071.

```
routine PRINT
        (  ref S: array[1..?S_MAX] of char,  // format string
           variadic ref V: array[1..?V_MAX] of  general);
   declare
    (   Sn: integer,     // count through S
        Vn: integer);    // count through V

   macro  SEND(F, T) = "
        Vn := Vn + 1;
        if Vn > V_MAX do Error() endif //report an error...
        F(force(T): V[Vn]));
        stopswitch; "

   Sn := 0     // initialize S scan counter
   Vn := 0
   while Sn <  S_MAX do
        Sn := Sn + 1;
        if S[Sn] ne $% do PUTC(S[Sn]); loop endif;
        Sn := Sn + 1;
        switchon S[Sn] into
                case $I: case $i:
                        SEND(PUTI,integer)
                case $L: case $l:
                        SEND(PUTL,logical)
                case $B: case $b:
                        SEND(PUTB,boolean)
                default:
                        Error()  // Report an error
        endswitch;
   endwhile;
endroutine;
```

Code generated for PRINT

```
*
*
PRINT    CSECT                                                  000
         USING   *,15                                           001
         STM     14,12,12(13)                                   002
         B       Start                                          003
Dynsize  DC      F'80'                                          004
Static   DC      A(PRINTs)                                      005
Start    L       14,8(13)                                       006
         ST      13,4(14)                                       007
         LR      13,14                                          008
         A       14,Dynsize                                     009
         ST      14,8(13)                                       010
         L       4,Static                                       011
         BALR    3,0                                            012
         USING   *,3                                            013
         USING   4,PRINTs                                       014
*
*        get parameters
*
         L       5,0(1)                                         015
         BCTR    5,0                                            016
         LH      9,4(1)                                         017
*                                                               018
*                                                               019
         ST      9,76(13)                                       020
         LA      6,8(1)                                         021
*
*                Start of program code
*
         SR      7,7             Sn := 0                        022
         SR      8,8             Vn := 0                        023
*
*                while Sn < S_MAX
*
S.1      C       7,76(13)                                       024
         BNL     S.21                                           025
         LA      7,1(7)          Sn := Sn + 1                   026
         LA      9,0(5,7)                                       027
         CLI     C'%',0(9)       if S[Sn] NE '%'                028
         BE      S.4                                            029
         ST      9,72(13)            do                         030
         LA      1,72(13)                                       031
         L       15,temp.1       call PUTC(S[Sn])               032
```

```
S.2     BALR    14,15                                       033
        B       S.1             loop                        034
*
S.4     LA      7,1(7)          Sn := Sn + 1                035
*                               switchon S[Sn]
        SR      1,1                                         036
        IC      1,0(5,7)                                    037
        IC      1,80(1,12)                                  038
        IC      1,temp.6(1)                                 039
        B       *+4(1)                                      040
        B       S.19                                        041
        B       S.5                                         042
        B       S.11                                        043
        B       S.16                                        044
*
S.5     L       10,temp.2        case 'I':  case 'i':       045
        B       S.17                                        046
*
S.11    L       10,temp.3        case 'L':  case 'l':       047
        B       S.17                                        048
*
S.16    L       10,temp.4        case 'B':  case 'b':       049
*
S.17    LA      8,1(8)          Vn:= Vn + 1                 050
        C       8,0(6)          if Vn >  V_MAX              051
        BNH     S.18                                        052
        L       15,temp.5                                   053
        BALR    14,15            Error()                    054
*
S.18    LR      9,8                                         055
        SLA     9,2                                         056
        L       9,4(9,6)                                    057
        ST      9,72(13)                                    058
        LA      1,72(13)                                    059
        LR      15,10                                       060
        B       S.2              PUTx(force(T : V[Vn]))     061
*
S.19    L       15,temp.5       default:                    062
        B       S.2             Error()                     063
*           endswitch
*       endwhile
S.21    L       13,4(13)        return                      064
        LM      12,14,12(13)                                065
        BR      14                                          066
        END                                                 067
*            static storage section
```

```
PRINTs  CSECT                                                    068
temp.1  DC                                                       069
temp.2  DC                                                       070
temp.3  DC                                                       071
temp.4  DC                                                       072
temp.5  DC      V(Error)                                         073
temp.6  DC      2X'00',X'0C',6X'00',X'04'                        074
        DC      2X'00',X'08',14X'00'                             075
        END                                                      076
```

14.1.2 Sample Program TOP2

Our next example program finds the top two numbers, within a bound, from an expression computed from array variable values. If the bound is exceeded the function returns the values at that time and sets a flag to increase the bound on the next call. This example may not be as logical in function as the previous one, but the point should be clear. Considerably more computation could have been inserted to make the function of the procedure more logical, but the processing would be the same and the details unnecessarily repeated.

```
 1  function TOP2
 2        ( ref a,b,c: array [1..10] of float,
 3          ref n: 2 byte integer):
 4          array [1..2] of float

 5     declare
 6      (    T: array[1..2] of float initially
 7               construct(array[1..2] of float, [1..2]:0.0)
 8           BIGMAX: static float initially(1000.0)
 9           SUM: 8 byte float
10                    initially(a[1]+b[1])
11           flip: static boolean initially(false)
12           i: integer);

13     SETBIG(BIGMAX, flip);
14     check float_overflow;
15     for defined i := n  step -1 until i <= 0 do
16          SUM := 0;
17          for j:= 1 to n do
18             SUM *= + abs(a[i] * b[j] + c[j] * a[i])*0.5
19             if float_overflow do goto Bad endif
20             endfor;
21          test SUM > T[1] do
22             T[2] := T[1];  T[1] := SUM    // reset top 2
23          orif SUM > T[2] do
24             T[2] := SUM                   // reset 2nd number
25          endtest
26          if T[1] > BIGMAX and i ne 0 do  // exceeded?
27             flip := true                  // yes, terminate
28             break
29             endif
30          endfor;
31     resultis T;

32  Bad:      // overflow case
33     flip := true;
34     resultis construct(array [1..2] of float, [1..2] :-1.0);
35     endfunction;


36  routine SETBIG
37        ( ref Big: float,
38          ref flag: boolean)
```

```
 39      declare
 40          (CONST = 500.0);

 41      if flag do Big := Big + CONST + 500.0
 42                    flag := false; endif
 43      return
 44      endroutine
```

Code for TOP2 (and SETBIG)

```
*
*
TOP2    CSECT
        USING   *,15                Standard Linkage:      000
        STM     14,12,12(13)    save callers registers     001
        B       Start                                      002
Dynsize DC      F'104'          dynamic storage needed     003
Static  DC      A(TOP2.s)       static storage address     004
Start   L       14,8(13)        get stack top              005
        ST      13,4(14)        save old stack address     006
        LR      13,14           stack for this routine     007
        A       14,Dynsize      compute new stack top      008
        ST      14,8(13)        save it                    009
        L       4,Static        set static base            010
        BALR    3,0             set program base           011
        USING   *,3                                        012
        USING   4,TOP2.s                                   013
*           get parameters and initialize
        MVC     76(8,13),temp.1                            014
        LM      6,9,0(1)                                   015
        ST      6,72(13)                                   016
        A       8,28(12)                                   017
        A       9,28(12)                                   018
        L       6,16(1)                                    019
        LH      6,0(1)                                     020
*
        LA      1,temp.2                                   021
        BAL     14,SETBIG       SETBIG(BIGMAX,flip)        022
*
*       check float_overflow
*
        MVC     96(1,13),1200(12)                          023
        LA      1,84(13)                                   024
        LA      15,1412(12)                                025
        BALR    14,15                                      026
```

```
*
        LR      1,6                                                  027
        BCTR    1,0                                                  028
        SRA     1,2                                                  029
        AR      7,1                                                  030
        L       10,28(12)                                            031
        LR      11,6                                                 032
        SRA     11,2                                                 033
*                for defined i := n step -1
        LTR     6,6             until                                034
        BNP     S.8                     i <= 0  do                   035
S.1     SDR     4,4              SUM := 0                            036
        LE      6,0(7)                                               037
        SR      5,5              for j:= 1 to n do                   038
        B       S.3                                                  039
*                  SUM =: + abs(a[i]*b[j]+c[j]*a[i])*0.5
S.2     LE      0,0(5,8)                                             040
        AE      0,0(5,9)                                             041
        MER     0,6                                                  042
        LPER    0,0                                                  043
        HER     0,0                                                  044
        AD      4,0                                                  045
*
        TM      X'80',1212(12)  if float_overflow do                 046
        BO      S.10              go to Bad; endif;                  047
S.3     BXLE    5,10,S.2                                             048
*                                 endfor;
S.4     LE      0,76(13)                                             049
        CDR     4,0               if SUM > T[1] do                   050
        BNH     S.5                                                  051
        STE     0,80(13)           T[2]:=T[1]                        052
        STE     4,76(13)           T[1]:=SUM                         053
        B       S.6.1                                                054
*
S.5     LE      0,80(13)           orif SUM > T[2] do                055
        CDR     4,0                                                  056
        BNH     S.6                                                  057
        STE     4,80(13)            T[2] := SUM                      058
*                                  endtest;
S.6     LE      0,76(13)                                             059
S.6.1   CER     0,BIGMAX          if T[1] > BIGMAX                   060
        BNH     S.7                  and                             061
        LTR     6,6                 I ne 0   do                      062
        BNZ     S.7                                                  063
S.11    MVI     X'FF',flip        flip := true;                      064
        B       S.8               break;  endif;                     065
```

```
S.7     SR      7,10                                    066
        BCT     6,S.1                                   067
*                               endfor;
*
S.8     L       1,72(13)                                068
        MVC     0(8,1),76(13)     resultis T            069
S.9     LA      15,1480(12)                             070
        BALR    14,15                                   071
*                               Standard return:
        L       13,4(13)         restore stack          072
        LM      14,12,12(13)     restore registers      073
        BR      14               return                 074
*
S.10    MVI     X'FF',flip       flip := true;          075
        L       1,72(13)                                076
        MVC     0(8,1),TEMP.3     resultis              077
        B       S.9                  construct(array...) 078
*       END                       endfunction;          079
*
*       function SETBIG
*
*       R6 = A(Big)
*       R7 = A(flag)
*
SETBIG  STM     6,7,44(13)                              081
        STD     0,96(13)                                082
        LM      6,7,0(1)                                083
        TM      X'FF',0(7)       if flag do             084
        BZ      S.11                                    085
        LE      0,0(6)                                  086
        AE      0,temp.4          Big := Big + 1000.0   087
        STE     0,0(6)                                  088
        MVI     X'FF',0(7)        flag := true; endif;  089
S.11    LM      6,7,44(13)                              090
        LD      0,96(13)         return;                091
        BR      14               endroutine;            092
        END                                             093
*               static storage for TOP2
TOP2.s  CSECT                                           094
temp.1  DC      E'0.0,0.0'       Initial value for T    095
BIGMAX  DC      E'1000.0'        BIGMAX, initialized    096
temp.2  DC      A(BIGMAX)                               097
        DC      A(flip)                                 098
temp.3  DC      E'-1.0,-1.0'     array: [1..2] -1.0     099
flip    DC      X'00',3x'00'     flip, initially false  100
temp.4  DC      E'1000.0'        constant for SETBIG    101
```

END 102

14.2 Characteristics of the IBM-360

Understanding some of the later discussion on the 360 COL compiler, as well as the sample program code, requires some knowledge of the target machine. In this section we (very) briefly describe the machine to assist the unfamiliar reader. (A more definitive description of the issues examined here, as well as many other properties of the machine, can be found in the System/360 Principles of Operations [IBM publication GA22-6821] and System/370 Principles of Operations [IBM publication GA22-7000]).

The basic characteristics of the IBM-360 might be summarized as follows:

1. 8-bit byte addressable memory, with a possible address space of 16,777,216 bytes.
2. 16 (32-bit) 'general purpose' registers and 8 (64-bit) floating point registers.
3. A (64-bit) register , called the Program Status Word or PSW, contains the program counter and status information.

4. Many data-types including three classes of arithmetic: binary, floating point, and decimal. Binary integers are represented in two's-complement notation.

5. Large instruction set; relative or 'base' addressing is used.

6. Complex interrupt structure.

In addition to bytes, the 360 references memory in terms of 2-byte 'halfwords', 4-byte 'fullwords' and 8-byte 'double words'. We use the term 'word' to indicate fullword, as is common in 360 literature. Also, when referencing memory in one of these forms it is generally required that the address satisfy a boundary alignment (the address is divisible by the length). By stating that an object resides in one of these forms, we will imply that the respective boundary requirement must be met. (Saying that an item resides in 2 halfwords implies that it is 4 bytes in length and must be on a halfword boundary.)

The general purpose registers are used mainly for addressing, binary integer arithmetic, and logical operations. Although register zero cannot be used in the instructions for a memory addressing register, any of these registers may be used where a general purpose register is required. Registers one and two serve additional functions in a few instructions. The

floating point registers are used with the floating point operations - the type of instruction will distinguish the register as either general purpose or floating point. In this discussion, when the term 'register' is used alone it will imply 'general purpose register', unless otherwise distinguished.

The status information in the program status word (PSW) includes a 2-bit condition code used in conditional branching, two sets of interrupt masks, an interrupt code and the length of the present instruction. The condition code (CC) is set by arithmetic, compare, and test operations and represents four distinct possible states.

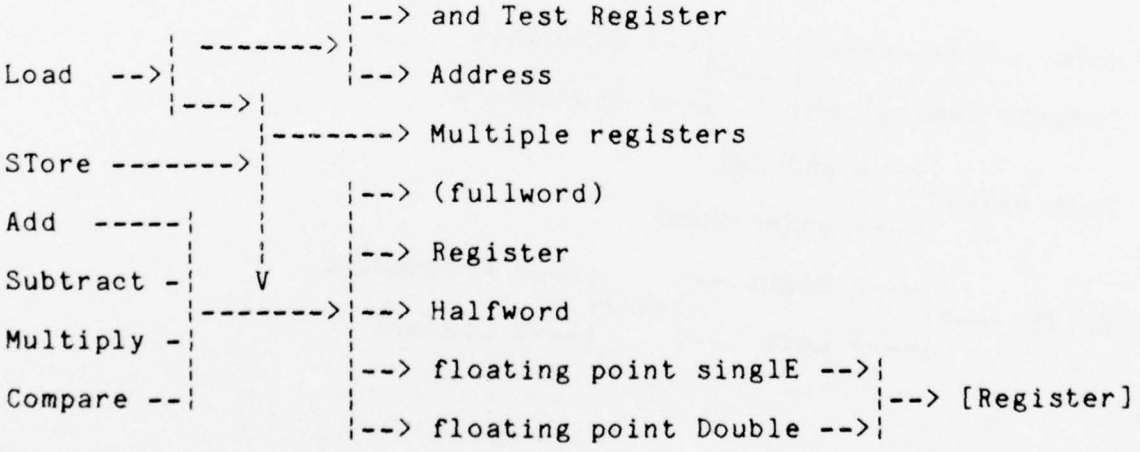
The machine datatypes correspond with the different views of memory. Binary integer instructions are defined for both fullword and halfword data. Arithmetic operations on this data always have one operand in a register (which is a fullword in length); halfword operands are automatically coerced into fullwords (and conversely). Similar to integers, floating point arithmetic operations have one operand in a floating point register (a double word in length) and length coercion is automatic. Floating point data may be in fullwords or double words. Instructions are available for decimal arithmetic, character operations, and character vector operations. Also, the primitives for other data types, such as interlocks, are provided.

All instructions address main memory with 'relative' or 'base' addressing. In an instruction, any memory address field consists of two parts: a 12-bit 'displacement' field and a 4-bit 'base register' field. The displacement is read as a positive integer between 0 and 4095; the base register is simply one of the general purpose registers. The memory reference, from an address field, is determined by adding the contents of the base register and displacement field plus any indexing. Another way to view this process is that all memory reference fields are indexed, sometimes double indexed, and an instruction has an available address space of 4096 bytes. Indirect addressing is not available and, as implied earlier, a register value of zero is used to indicate no base register (or no index register when used in the index register field). It should also be noted that address computation always results in a positive value with the upper 8 bits being set to zero.

Instructions may be one, two or three halfwords in length. One halfword instructions generally involve operations between two registers. Two halfword operations incorporate instructions between a single register and an indexed memory address (such as add), a group of registers and memory (such as load multiple), and data in the instruction and memory (such as compare immediate). Instructions that are three halfwords in length are

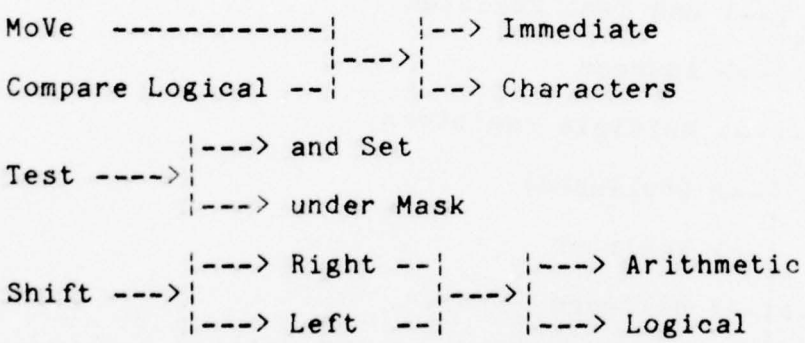
operations between two objects in main memory (such as move characters). Note that in discussion the base register is always included in the term 'memory address'; operations such as those between two items in storage may require two additional registers to maintain the base addresses.

We present a brief summary of some of the instruction set in the following tables. A mnemonic used in the code can be formulated by appending the upper case letters while following a line in the graph, along any direction indicated. The terms along the path basically describe the function of the operation.



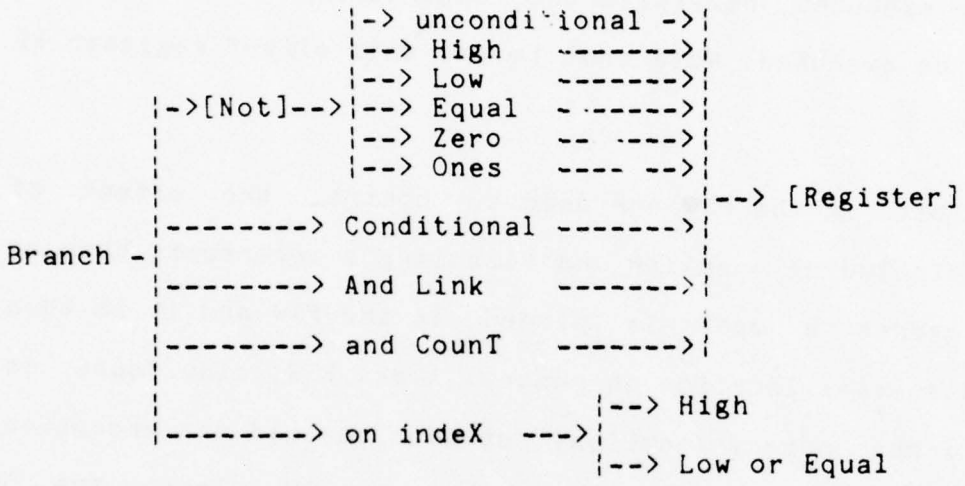
Operations ending in 'Register', e.g., AER, indicate a register to register operation - both registers are the same type: floating point or general purpose. Memory addressed with

codes ending at (fullword) and below allow for an index register in the operation. Multiple register operations, like store multiple (STM), specify moves between a sequence of registers and storage locations and the register sequence 'wraps around' at zero. STM 14,0,4(1) would store registers 14, 15 and 0 (in that order) in 3 consecutive words starting with address 4 plus the contents of register 1. The load address operation puts the indexed address directly into a register, for example LA 1,W would put the address computed from W into register 1 - note W is written D(X,B) with D the displacement value, X the index register and B the base register.



Operations like move characters (MVC) involve two storage operands, MVC D1(L,B1),D2(B2) moves L bytes from address D2(B2) to D1(B1). Immediate data instructions have an 8-bit data field in the instruction, for example CLI X'0A',D(B) will compare the

byte contents of D(r) with 10. (The notation X'dd' indicates a hexadecimal constant whose value is dd.) The operand of shift is like the load address instruction (the value of the address field is used for the shift count).



The basic branching operations, those above 'conditional', are all forms of the branch conditional operation (BC and BCR). In this instruction the first operand is a four bit mask used in testing the condition code, and these special *mnemonics* are used for readability (BNZ W as opposed to BC 7,W). Branch and link (BAL and BALR) sets the next instruction address in the register indicated by the first operand and branches to the second operand address. Branch and count counts a register down by one and fails to branch when zero is reached; branch on index instructions

involve three registers outside of a base. When executed, the instruction BXLE R1,R2,D(B) will add the contents of register R2 to R1 and compare the result with R2+1, if the result is less than or equal to the contents of R2+1 then the branch to address D(B) will be executed, otherwise the instruction following the BXLE will be executed. Note that to get this effect register R2 must be even.

Status bits in the PSW are used to control the effect of interrupts, including enabling and disabling a interrupt. When an interrupt occurs a code is placed in the PSW and it is then stored in a special location in memory. A new PSW, also found in a distinguished memory location, replaces the old and execution proceeds at the address indicated in the new PSW. There are 5 major classes of interrupts, of which input/output is only one; effects of program operations (such as overflow) is another class of interrupts.

The machine highlights we have provided above only touch on the many characteristics and instructions of the IBM-360. As stated in the introduction of this section, the intention was to assist the reader in understanding the sample program code and text that follows.

14.3 Data Representation

Representing COL data types is straightforward on the 360, with access and some coercion frequently done in a single instruction. In this section we examine the representations for COL. Unless otherwise specified, data types specified as a particular machine representation, such as a fullword, must be located at a respectively aligned address. We begin by considering the basic types.

14.3.1 Basic Types

Integers are represented naturally as fixed, either halfwords (2 bytes) or fullwords (4 bytes), depending on the declaration. The access instructions available for each form provide an automatic coercion between lengths. In the program PRINT, S_MAX is initially fetched (line P017) from a halfword and then stored in a fullword temporary (line P020); V_MAX is retained in a fullword (line P051). When specified, a byte is used for unsigned integers, but this is not generally recommended since most uses require a conversion step.

Variables BIGMAX and SUM in the TOP2 program represent single and double precision uses. BIGMAX is kept in static storage (at T098) and SUM remains in a double word register and

is used from there (lines T036, T045, T050, T053). As with the integers, the floating point operations provide an automatic length coercion (line T045).

The character array S in PRINT has been used with some character operations (lines P028, P037). Data movement operations (such as T077) are also frequently used. The character code may be specified at compile time and it should be noted that the "machine" character code for the 360 is EBCDIC. Easy conversion between character codes can be done with a translate instruction but some processors require EBCDIC for internal character functions such as binary conversion and I/O.

Boolean data is easily managed in a byte with zero representing false and all ones representing true; for example, see variable flip in TOP2 (lines T064, T084). Logical data simply assigns one byte for each 8 bits in the basic type size. Logical and boolean are coded almost identically; in some special cases logical may be used with register operations.

Interlocks are represented like boolean with all zeroes for locked and ones for unlocked. Represented in this form a single instruction (test-and-set) is used with an interlock to test and lock (if unlocked). Since this instruction sets the machine condition code to reflect the state of the interlock, coding is quite straightforward.

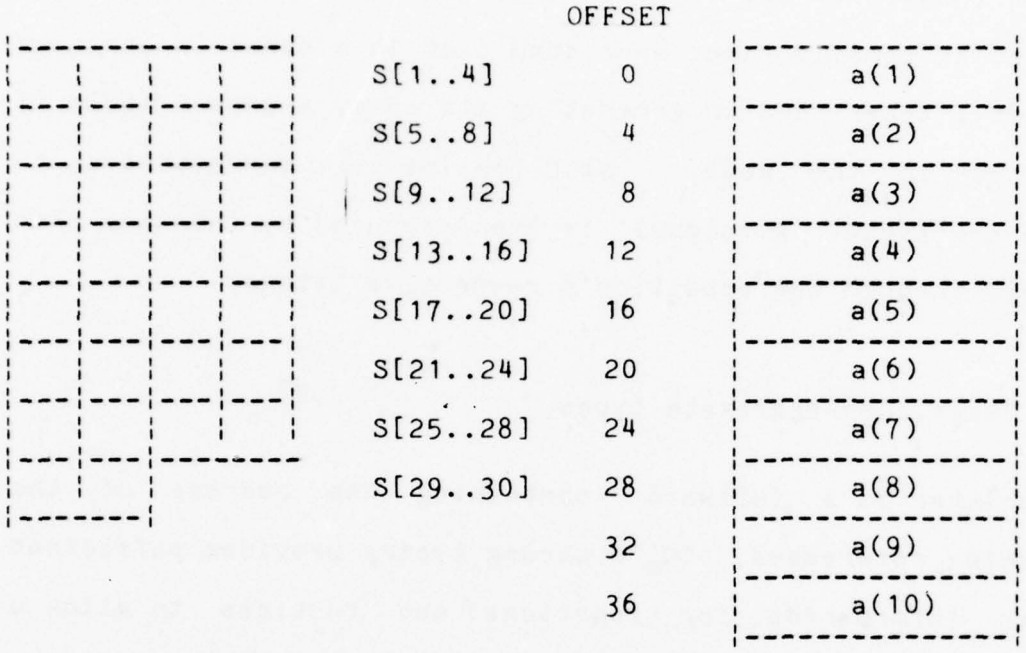
Zahn conditions are represented in the code produced for the procedure. At compile time, each condition in a block is assigned a distinct integer used in generating its code. A branch table is constructed in the code, with the integers corresponding to jumps in the table. A 'signal' is then produced by indexing into this table through the condition's respective integer.

14.3.2 Other, non-aggregate types

A pointer is a fullword containing the address of the object being referenced. COL's strong typing provides sufficient parameter information for functions and routines to allow a general reference to be done through a pointer containing its address.

14.3.3 Aggregate Types

Arrays of basic type elements are stored as a vector of consecutive storage units, whose size is that of the basic type, with the lowest indexed element at the lowest address. A storage layout for character array S (with S_MAX = 30) and float array a in TOP2 is shown in the figure below. (These and all other storage diagrams represent four bytes across with consecutive addresses increasing left to right and top to bottom. The 'offset' is that from the address of the first byte.)



ARRAY S ARRAY a

Note that an array of 10 fullword integers or pointers would be stored in the same form as above and an array of boolean, 8 bit (or less) logicals, and interlocks would be stored (and addressed) like S.

Array access is done with both indexing and pointers. The terms 'adjusted base address' and 'index multiplier' are common in discussing array addressing and help in describing both addressing approaches. The 'index multiplier' is the number of bytes between consecutive array elements; for S this is 1 and for

a it is 4. Subtracting the product of the array's lowest index and the index multiplier from the address of the first element in the array produces the adjusted base address. Array variables in a loop commonly have the same multiplier and thus share the the same offset from the base for any index; for example b and c in TOP2 share the same index (T040, T041). When addressing is random, the product of the index value and its multiplier is added to the adjusted base to obtain the address of the data element. An example of this is provided in PRINT when V is referenced (P056, P057). Arrays used only in this way need only the adjusted address (T017, T018).

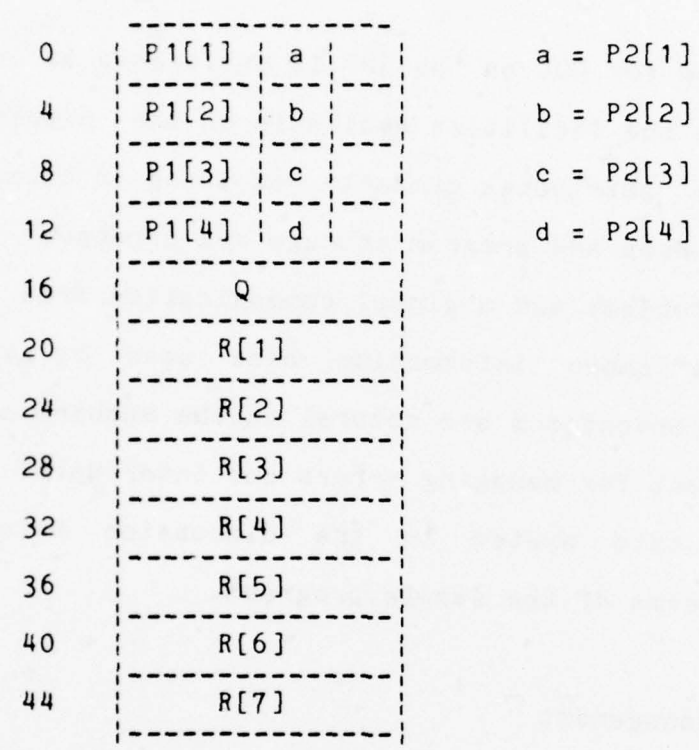
Pointer addressing is done when a common index is not shared and a register is available. Array a in TOP2 is used as a pointer addressed array (T037). Generally, indexing is preferred to pointer addressing, since there usually is a corresponding loop index or an extra register is not available.

Note that arrays of basic types have all elements properly aligned if their first element is aligned and the arrays are packed (with the exception of packed packed).

Structures are stored as consecutive substructures with alignment being adjusted as necessary, as in the example below. Note that this is not parallel; the parallel structure would be more compact and should perhaps be the default.

```
declare
        (T is structure
                (P: array [1..4] of structure
                        ( P1: 2 byte integer;
                          P2: boolean );
                 Q: integer;
                 R: array [1..7] of integer);
               TA is array [1..2] of T;
              V: T
              VA: TA);
```

Storage layout for V



It is interesting to note that the multiplier for both P1 and P2 is 4 in the above example, even though they are variables of different lengths. Accessing any value in V is identical to what we have seen before for the corresponding primitive data type, with offset adjustments coded in the instruction.

14.4 Organization of the Run Time System

The run time system for COL on the 360 is envisioned as one that attempts to utilize the facilities available on the machine while minimizing those attributes commonly resulting in clumsy programming constructs. Data and program storage are arranged to simplify addressing problems and a global communication area is established to provide a common information base used by all procedures. Linkage conventions are natural to the machine and provide a good environment for managing errors and interrupts. We shall examine parts of this system in the discussion below, providing examples in terms of the sample programs.

14.4.1 Global Storage Management

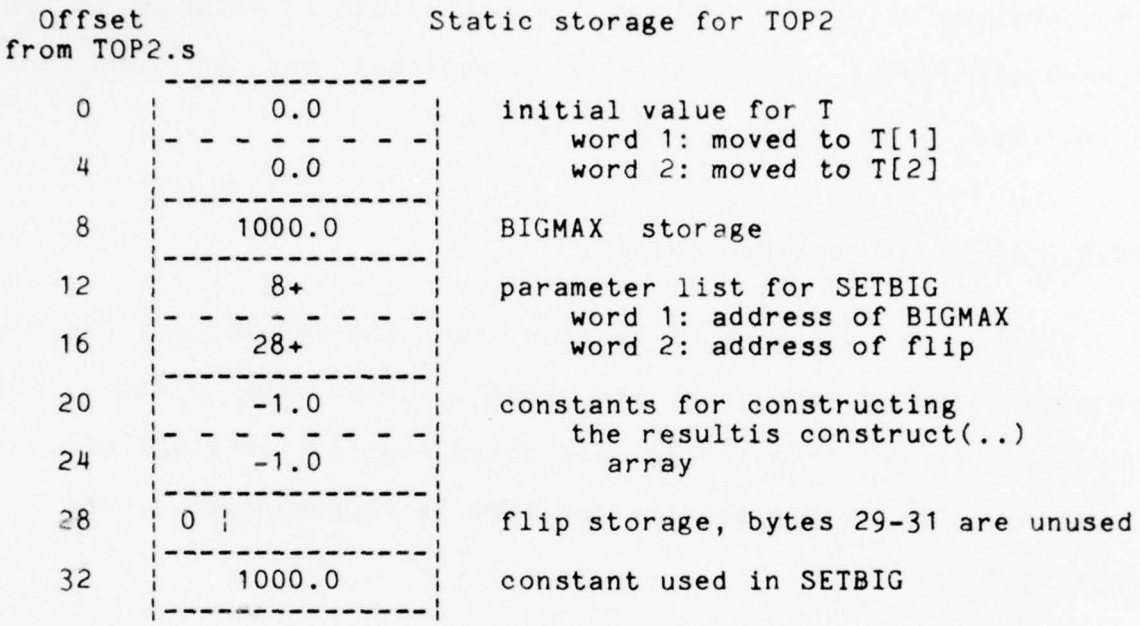
Storage will be divided into three pools: static, stack, and controlled. Program code and static variables reside in static storage while dynamic and allocated variables are placed in either stack or controlled storage. Initially, we shall view storage divided into two segments, one containing static storage and the other used for the stack and controlled storage.

Stack storage usage follows first-in-last-out order and is allocated and released in blocks by a procedure, for its dynamic variables, temporaries, and other information. Controlled storage

is obtained using the "allocate" function and is returned by the "free" statement. Any of several algorithms may be used for this. See, e.g., section 2.5 of [Knuth]

14.4.2 Local Storage Management

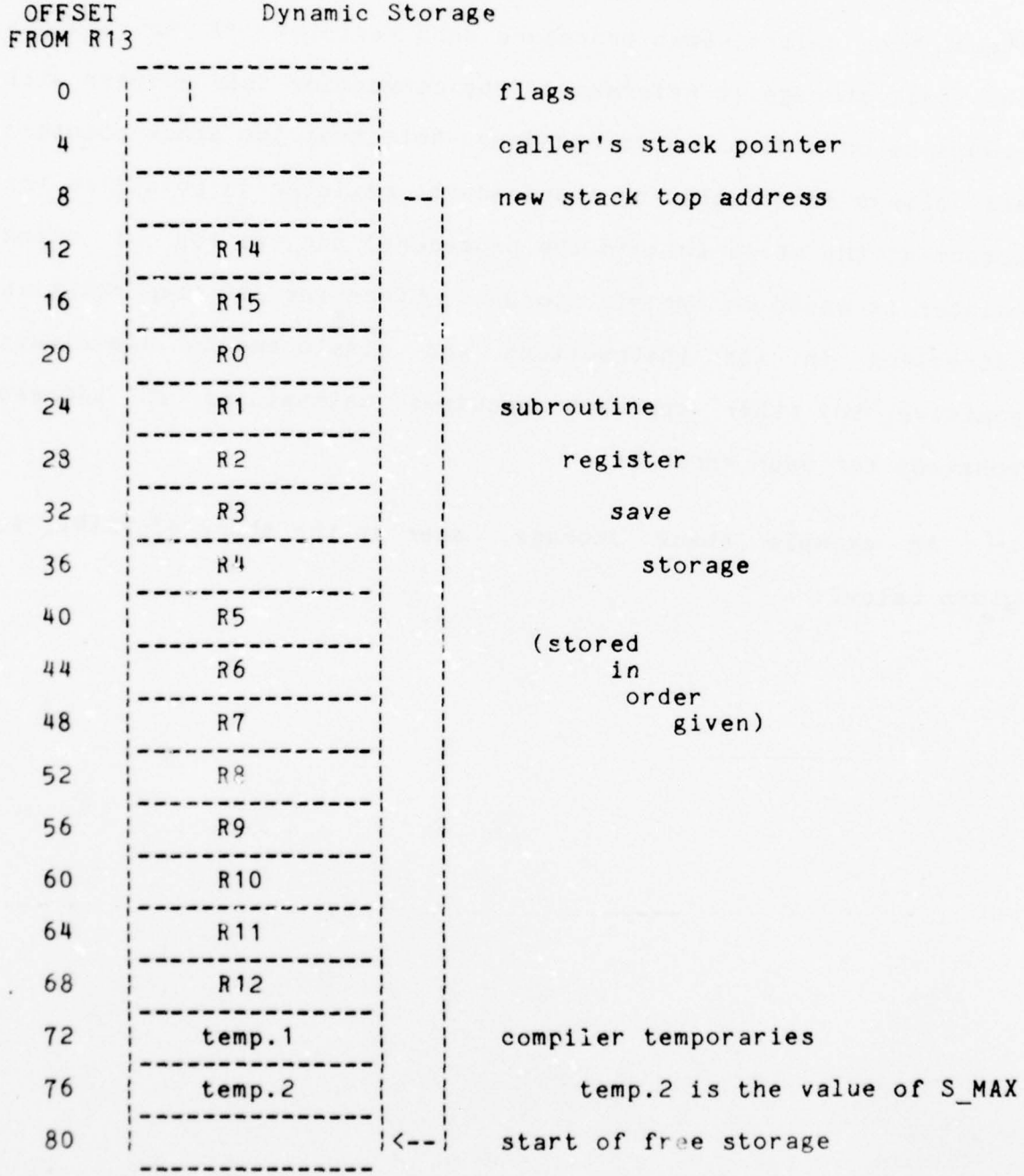
There are two control sections for each module: one for the program code and one for its static data. The static data includes program variables (of class static), data and address constants. The static section for TOP2 is representative and is displayed below.



Stack storage mainly behaves like a 'stack' between procedure calls. Upon initiation a procedure allocates all necessary stack storage, which can be determined at compile time. When the procedure returns, it releases any stack storage used. This policy appears best suited to the nature of addressing and data alignment requirements for the 360. Maximum reuse of temporary storage will always be exercised to minimize the space requirements for this area. Within a procedure, assigning storage by largest alignment first (double words, fullwords, halfwords, then bytes) is usually best.

The stack is essentially a structure of all the dynamic data items. When called, each procedure sets register 13 to address the stack storage it references; by convention this address will always be on a double word boundary. Note that two stack pointers are always established by a procedure: register 13 points to the bottom of the stack (within the procedure) and a top of stack pointer is saved in dynamic storage. Since the 360 uses relative addressing in its instructions and displacements are always positive, any other approach requires maintaining an address constant for each variable.

An example stack storage, seen at the start of PRINT, is given below.


OFFSET FROM R13
Dynamic Storage
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
68
72
76
80
R14
R15
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
temp.1
temp.2
flags
caller's stack pointer
new stack top address
subroutine
register
save
storage
(stored
in
order
given)
compiler temporaries
temp.2 is the value of S_MAX
start of free storage

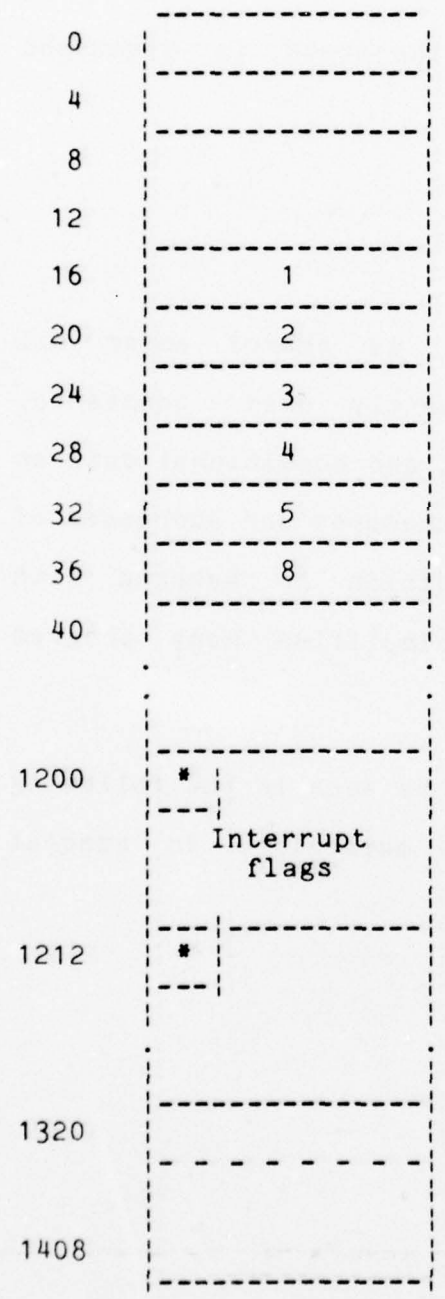
The process involved in allocating the stack is discussed further under linkage.

14.4.3 The Global Communication Area

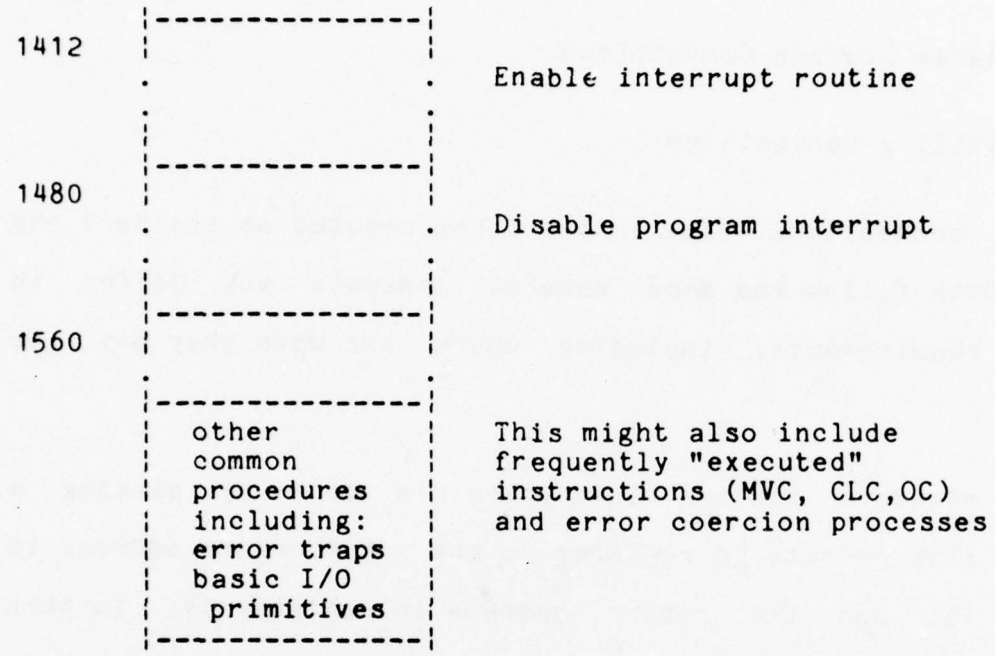
The Global Communication area is shared among all procedures. This area includes frequently used constants, storage management information, interrupt and conditional data as well as the code for some basic procedures and addresses of frequently referenced programs. In addition to keeping with COL's philosophy, this construct simplifies many program requirements.

A possible outline of this area can be seen in the following diagram. The address of this area is maintained in general register 12.

Global Communications Area



(continued on next page)



The TOP2 program references this area at several points. Constants are used in lines 017, 018, and 023. Interrupts are enabled through a procedure in this area (lines 024-026) and later disabled (lines 069-071). Also an overflow flag from this area is tested (line 046).

14.4.4 Program Linkage Conventions

14.4.4.1 Calling Conventions

Call conventions come in two forms denoted as standard and simple. Both follow the same general schemata but differ in specific requirements, including where and when they may be used.

A standard linkage invocation is done by placing a parameter list address in register 1, the subprocedure address in register 15, and the return address in register 14. In this case, the subprocedure is required to establish its own environment (adjust the stack, etc.). For example PRINT calls PUTC using standard linkage (lines 031-033).

When a procedure call occurs, establishing the environment takes several steps; these are seen in lines 001-013 of the TOP2 code. As noted in the code the caller's registers are saved (line 001) in his environment and the stack is modified for the new environment (lines 005-009), with the old stack address saved in the new one (line 006). Note the the constant Dynsize (line 003) which contains the number of bytes of stack storage needed, known at compile time; the constant Static (line 007) is the starting address of static storage for the procedure. Finally, observe

that register 3 serves as a program base address and register 4 is the static data base address (lines 012-013).

A simple linkage call is similar; for example, in TOP2 the call to SETBIG and the enable routine are simple (lines 021-022, 024-026), but a simple procedure call does not require the called procedure to establish its own environment (but it must preserve any of the caller's environment, such as registers). This may be used by private routines within a module and is established when indicated by the compiler.

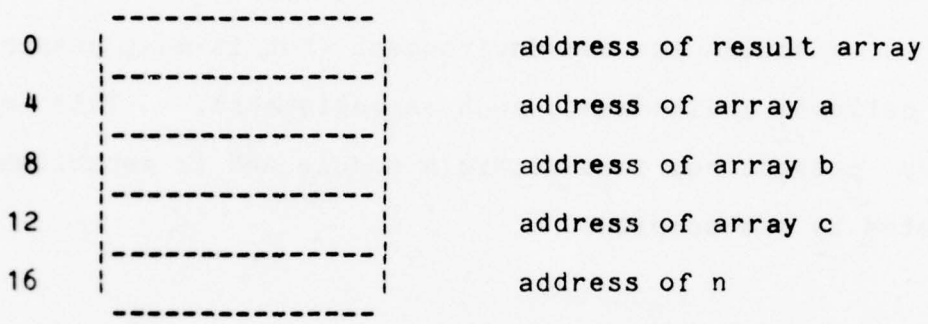
14.4.4.2 Parameter Lists

The arguments of a procedure, routine or function, are obtained from the parameter list. This structure provides the information necessary to access and use the parameters.

Most commonly the parameter list is simply a vector of pointers. Each pointer contains the data address for a parameter, and they are listed in the same order as specified in the procedure header. For a routine, the first element of this vector corresponds with the first argument, the second element with the second, etc. In the example for the static storage for TOP2 above, the two word vector starting at offset 12 is a sample parameter list for SETBIG. A function parameter list is

constructed in the same manner, except that the first element will correspond with the result. For example, a storage layout for a parameter list for TOP2 is seen in the following diagram.

Parameter List for TOP2

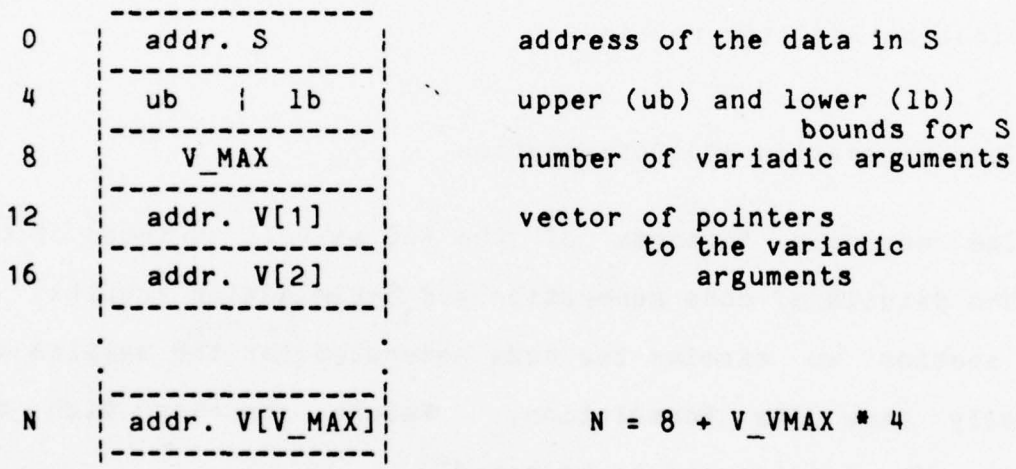


In some situations the address of the data is not sufficient. In particular, this is the case with flexible arrays and variadic arguments. In each of these cases the parameter list is extended by adding additional data.

When flexible arrays are used the upper and lower bounds of the array are added to the parameter list, following the address of the array. Two halfwords are used to store the array bounds, with the upper bound preceding the lower one. S is a flexible array in PRINT and a model parameter list for PRINT follows.

Variadic arguments are managed by a simple extension to the parameter list. Following the non-variadic parameters on the list, an integer word containing the number of variadic arguments followed by a vector of their addresses is appended. A parameter list for the sample program PRINT is of the form indicated in the following diagram.

Parameter list for PRINT



This additional parameter information is constructed only when absolutely necessary, and generally as much as possible is done at compile time. This is true with parameter lists as well; if most of a parameter list can be completed at compile time it will be, and saved in static storage.

14.4.5 Interrupts and Condition Handling

Interrupts are managed by having a small trapping routine for each class of interrupt that passes control on to another routine based on the specific interrupt code. The processing routine should be indicated by the General Communication Register. Enabling and Disabling interrupts and establishing routines to manage a specific interrupt are done thru the General Communication Register routines.

14.5 Code Generation and Optimization

The numerous features of the 360 as well as those of COL make the details of code generation and optimization complex. In this section we examine the code generated for the samples and generally view its formulation. Before starting with the samples a few points need to be noted.

The optimizations used below do not cover all known optimization strategies, nor do the examples represent best case situations for those applied. The major point of this discussion is that COL statements can be reasonably compiled on the 360; but since the actual code produced for any statement or part thereof is not independent of those surrounding it, some optimization is applied to reflect the compiling process. It should be noted

that at some places the code was deliberately unoptimized so other situations could be described.

14.5.1 Sample Program PRINT

Before starting the code generation process for PRINT, it is necessary to expand the macro. Once expanded the program will look as follows:

Macro expansion of PRINT

```
 1  routine PRINT
 2          (  ref S: array[1..?S_MAX] of char,  // format string
 3             variadic ref V: array[1..?V_MAX] of  general);
 4      declare
 5       (    Sn: integer,     // count through S
 6            Vn: integer);    // count through V

12      Sn := 0     // initialize S scan counter
13      Vn := 0
14      while Sn <  S_MAX do
15            Sn := Sn + 1;
16            if S[Sn] ne $% do PUTC(S[Sn]); loop; endif;
17            Sn := Sn + 1;
18            switchon S[Sn] into
19                    case $I: case $i:
20.1                   Vn := Vn + 1;
20.2                   if Vn > V_MAX do Error() endif;
20.3                   PUTI(force(integer:V[Vn]))
20.4                   stopswitch;

21                    case $L: case $l:
22.1                   Vn := Vn + 1;
22.2                   if Vn > V_MAX do Error() endif;
22.3                   PUTL(force(logical:V[Vn]))
22.4                   stopswitch;

23                    case $B: case $b:
24.1                   Vn := Vn + 1;
24.2                   if Vn > V_MAX do Error() endif;
24.3                   PUTB(force(boolean:V[Vn]))
24.4                   stopswitch;

25                    default:
26                           Error()  // Report an error
27            endswitch;
28      endwhile;
29  endroutine;
```

Following standard linkage (000-014), the parameter information is obtained (015-021). Generally the address of a

parameter is obtained and with arrays it may be adjusted for indexing (line 015 obtains the address of S, and line 016 adjusts it for indexing); depending on the register situation, this address may be saved in dynamic storage or in a register -- in this case a register was assigned. Note that the optimization process simplified the adjustment by generating a fast instruction to do the 'subtract 1' operation (such optimization is called good local code generation).

The variable S_MAX (obtained in lines 017-020) breaks the general rule and establishes a new one: if the value of a variable is not modified and its storage requirements are less than or equal to a fullword (the size of an address constant), then save the value of the variable. Finally V and V_MAX are obtained using a third rule specific to variadic parameters: retain the length field address in the parameter list (line 021). Since the list of pointers follows this word, this address is the adjusted address of the array and the address of its length as well. (This observation allows one register allocation for both of the variables).

The initial zero settings for Sn and Vn, lines 12-13, are easily generated by self-subtraction (lines 023-024). Note that for variables not in registers an exclusive-or operation will

produce the same effect. Lines 14-28 contain the while-do iteration loop and are coded in lines 024-063. Lines 024-025 compare Sn to S_MAX and exit to line 064 when Sn is not smaller. Iteration takes place at the loop statement (line 034) and at the endwhile (line 063). In this case the peephole optimization of branch chaining and cross jumping merge common code preceding the iteration. Note that repeat-until iteration would be coded in a similar manner, but the test, lines 024-025, would follow line 063 and would branch to line 026 on an unsuccessful test.

Another example of good local code generation appears on line 026 where Sn is incremented (line 15). If Sn were not in a register at that time, a load and possibly a store would have to enclose the operation.

The conditional on line 16 is done by indexing S (line 027) and testing the character against $% (line 028). If the characters are equal, the code for the do arm (lines 030-034) is skipped (line 029). Within the do arm routine PUTC is called (lines 030-033). Here evaluating the address of S[Sn] for PUTC's parameter list was redundant with that computed on line 027.

The switch block (lines 18-27) is coded in lines 036-063. In this example, as in most switch codings, the case selection is done through a branch table. To select the proper branch, the

character S[Sn] is translated (lines 036-039) into a branch table offset that indexes the jump (line 040) through the branch table (lines 041-044). For this example the character, S[Sn], is first translated into a number from 0-26 (line 038), using data available in common storage; this number is then used to index a table in static storage (line 039) to produce the proper branch index. If the case branches are consecutive integers, the translation tables are unnecessary: the offset is computed directly from the case number; also, if the translation cannot be done by a table index or simple computation then a test-orif construction is used.

The body of the switch statement has been greatly simplified through optimization. First compare program lines 20.1-20.4 with the code lines 050-061. Program lines 20.1 and 20.2 are coded like lines 15 and 16. To obtain the index for V, Vn must be multiplied by 4 to compute the offset, and for this, the code generally uses a shift (line 056). The next step is to obtain the address of the data indicated by the pointer V[Vn], which is done on line 057; this sometimes takes two steps, but good local code generation would always result in one instruction. This produces the indirection, and now the data can be used as an integer (resulting from the force). The remaining steps to call the subroutine (lines 058-061) are nearly identical to the

previous call (lines 030-033) with cross jumping used at line 061 to save an additional operation. The only difference (line 060) is discussed below.

All case arms (lines 20.1-20.4, 22.1-22.4, 24.1-24.4) can be constructed in this manner and while still in an internal representation, will be. In this representation, a standard routine or function call is represented as two abstract operations: one corresponding to obtaining the function address, and the other corresponding to calling the procedure just addressed. Although usually these are done in sequence, code motion has merged with the common paths of the three arms and shifted the obtain address portion of the operation to the top of the case arms (lines 045, 047, 049). At this point the routine address is assigned to a temporary register and then moved to register 15 just prior to the call (line 060). Thus the majority of code for all three cases is merged together. The default case is handled as a separate arm generating calling code as seen previously.

The only remaining code follows the while loop and is that of a standard return.

14.5.2 Sample Program TOP2

The sample program TOP2 starts like PRINT with standard linkage (lines 001-013) followed by initialization and obtaining parameters (lines 014-020). T is initialized to a constant in static memory (line 014), with all other initializations being done at compile time. The code for initializing SUM was found to be useless and consequently deleted; if this were not the case, it would follow line 018.

Excluding n, all parameters are obtained with the general parameter rule. Parameter n follows the criteria for the second rule and thus its value is obtained. Note that several loads have been merged into one (line 015) by combining adjacent field moves; also the storing of n was later found to be useless, and thus deleted.

The call to SETBIG is done with simple linkage (lines 021-022) with the parameter list being prepared at compile time. The check statement (line 14) produces a simple linkage call to the enable procedure (lines 024-026) in common storage, after constructing a parameter list (line 023). Optimization converted the SETBIG call to a 'local' one.

The outer for loop code (lines 15-30) encompasses lines 027 through 067. With this loop, counting is used for iteration control. To do this, first a counting register is initialized (code deleted since it was redundant with line 020); then an initial boundary test is made (lines 034-035). The counter is decremented and tested (line 067) and the loop repeats (to line 036) if not finished. The defined i happens to be equivalent to the counter and is thus made to be an alias.

Iteration control in the inner loop uses the loop-indexing method available in the hardware. The increment and bound registers are set up (lines 031-033), the index is set to its initial value less the step size (line 038), and the loop proceeds by stepping and testing the index (line 048). The initialization code was moved out of the outer loop since it was loop invariant. Also, since j is an induction variable, its most frequently referenced use (j*4) is used for the index with the step and test being adjusted appropriately.

Nearly all forms of for-iteration will be implemented with one of these basic plans, the choice depending on the number of registers available and how the loop uses the index. Discrete lists may be unrolled, but the general rule would be to construct a table of values and then index through it.

The first step of the inner loop is to evaluate the expression on line 18. The array a is accessed with pointer addressing (line 037), which in this case is preferable since it can be kept in a register and no other variable uses the same index. The pointer is adjusted when the outer loop steps (line 066). Note that the access to A[i] was loop invariant and thus moved out of the inner loop. Terms b[j] and c[j] are accessed with indexing (lines 040-041). Factorization has been applied to the expression, a[i]*b[j]+c[j]*a[i], producing the simple expression (b[j]+c[j])*a[i] which is computed on lines 040-042. Following the absolute value function, that is in-line coded (line 043), division by 2 (line 044) is done to produce the multiplication by 0.5. The resulting value is then accumulated in SUM (line 045).

If floating point overflow occurred, it was trapped and a flag was set in common storage. The if statement (line 19) tests this flag (line 046), and if on, branches out of the loop (line 047).

A test follows the loop (lines 21-25). Checking the first condition (line 050) requires an explicit length coercion (line 049), and if false, a branch to the second condition test is made (line 051). The arm follows the conditional branch (lines

052-054) ending with a branch beyond the endtest. The fetching and coercing of T[1] is redundant with that needed for the test code and is thus deleted; the same is true of lines 049 and 060, and it is observed during peephole optimization.

The general pattern of testing one condition and, if false, branching to the next condition test or, if true, branching beyond the test statement (after the arm code), is used for all test statements. In this light, the 'otherwise' clause can be viewed as an always true test.

The if-conditional is evaluated after the test (lines 059-063) and if true a break is made (line 065). When the loop is terminated, the function value is set to T (lines 068-069), a procedure to disable traps set by the program is called (lines 070-071), and the procedure returns (lines 072-074). If the function branched to Bad (line 075), the result would be set from the constructed constant (lines 077-078), and the same return would be made.

The coding for SETBIG follows the patterns previously discussed, except that by using simple linkage, it does not need to establish a new environment, and only saves those registers that are absolutely necessary. It should be noted that the compiler has folded the two constants together so that only one addition (line 087) is necessary.

In closing, the reader is reminded that the main goal of this section was to demonstrate that code translation for COL was possible for the IBM-360. Since translation is always done within the context of a program, some optimizations were applied to make the result more representative. At times optimizations have been deliberately ignored, or used one place and not another, in order to discuss translation for a larger set of contexts.

[Knuth]
Knuth, Donald E., "The Art of Computer Programming - Volume I," Addison-Wesley, 1969.

Chapter 15

Security Analysis of the COL

In addition to studying the compilability of COL programs, we have studied the security characteristics of programs written in COL. We studied two aspects of program security, the data access control mechanisms and the provability of programs. A data access control mechanism is a mechanism which grants access to a collection of variables for some procedures while inhibiting access by other procedures. The design of a language impacts on the provability of programs written in it. To find those language constructs which make proofs difficult, we have made a first attempt at proof rules for COL statements.

One way to help attain program security is to guarantee that a particular program segment can not be affected by other portions of the program. This guarantee is implemented by limiting access to the variables referenced by the program segment. The COL has four characteristics which enhance this kind of data access control:

- encapsulated data types
- scope rules
- separate compilation
- strong type checking

The COL's scope rules and strong type checking are more stringent than those used in PASCAL. Functions and routines passed as arguments to other functions and routines must agree in arguments and results in the COL. (This is not checked in PASCAL.) Further, the COL permits the user to override the type checking rules by the "force" mechanism when necessary. These violations are explicitly noted by the compiler and flagged in the listing. In PASCAL trickery is used when the type of a datum must be changed. The scoping rules for the COL are more stringent than those for PASCAL. A COL procedure has access to those static variables defined in any procedure whose body contains the declaration of the procedure and those dynamic variables defined directly in this procedure body. On the other hand a PASCAL function or procedure has access to all variables declared in any function or routine whose body contains the declaration of the current function or routine. However, the scoping rules for neither PASCAL nor the COL are sufficient for the needs of secure programs. A "defines" and "uses" list should be added to each function and routine declaration as in MODULA[Wirth]. These lists would indicate which routines change or use which variables. The compiler can use these lists for optimization, for providing the programmer a table of routine interactions, and for controlling access to variables by routines.

Unlike PASCAL, the COL has been designed to allow separate compilation. Each module is compiled separately in the COL. Each module specifies which other modules it accesses or uses. Furthermore, the specifications for each routine in each module are recorded in only one place. These features limit the access to variables in other modules and guarantee the compatibility of declarations in intercommunicating modules. The variables, functions, and routines which can be accessed by other modules are listed in the module declaration. Hence the compiler (and user) can determine which other modules can access particular variables. Since each declaration is stated in only one place, there can be no compatibility conflicts introduced by changing a particular module and forgetting to update the declaration in other modules. Each module searches for declarations in the modules it has access to. Note that the COL module features require some form of editor and data base to guarantee that the proper object and source files are searched during linking and compilation.

Encapsulated data types give the strongest access control mechanism available in the COL. Encapsulated types are a generalization of structures with two added features. First, functions and routines can be defined within the structure. These functions and routines can access any field or variable in the

structure. These functions and routines should be thought of as operations on the structure as an atomic unit. Second, the declaration of the structure may specify that any particular field may or may not be accessed outside the functions and routines defined in the structure. With these features the programmer may hide the internal structure of the encapsulated data type from all users of variables of this type. This enhances the modifiability of the program and inhibits the use of variables and fields by unauthorized routines or functions.

The second component of our security analysis was to investigate the provability of programs written in the COL. The provability of programs is a function of the language structure, the programmer's style, and the task being programmed. We have investigated the effects of language structure on the provability of programs. Since this was a modest study of provability we have chosen to generate proof rules for some of the statement forms of the COL and to determine which statement forms have proof rules which are difficult or impossible to state. To make this task easier, we compared the COL with the axiomatization of PASCAL[Hoare-73]. Wherever the COL and PASCAL agree we will refer to the PASCAL document for proof rules and discussion.

The declarations and arithmetic structure of PASCAL and COL are similar. The COL is different from PASCAL in that the COL has "size" attributes for types, variable declarations can include an "initially" attribute, and the COL contains encapsulated data types. The size attribute introduces little complexity, since the compiler checks for size compatibility. A proof is needed only if the object computed will not fit into the destination. This proof is a simple range check that the value fits into the range indicated by the size. The "initially" attribute is equivalent to a declaration and assignment. This introduces no added complexity to proof rules and will not be further discussed.

Encapsulated data types, as implemented in the COL, are handled by name scoping. If F is a routine defined in an encapsulated data type T and V is a variable of type T then V.F(A,B) is equivalent to F(V,A,B) where the name F is disambiguated by the scoping rules of the encapsulated type T and the first argument V is called by reference. Hence encapsulated data types add no complexity to the proof rules.

The statement forms of the COL are as follows:

```
E := E
E *= <infix-op> E
E(E , ... )
if E do S; ... endif
```

```
unless E do S; ... endunless
test E do S;... orif e do ... endtest
while E do S; ... endwhile
repeat S; ... until E
for ID := E ... endfor
break
loop
goto ID
ID: S
return
resultis E
<empty>
switchon E into S; ... endswitch
<case> : S
stopswitch
upon ID or ... leave S; ... do S; ... endupon
signal ID
region E do S; ... endregion
retry
lock E
unlock E
failing SD; ... failhere SD; ... endfail
fail
fail finishing E
begin SD; ... end
{SD; ... }
```

Some of these statement forms have semantics identical to those of statement forms in PASCAL. Hence the proof rules will be the same for both languages. These statement forms are:

```
E := E
E(E , ... )
if E do S; ... endif
goto ID
ID: S
begin SD; ... end
{SD; ... }
```

Axioms for the use of "goto", as found in [Clint], carry over to PASCAL and the COL. Specifically, associated with each label ID

there is a predicate which must be true each time the label is reached either by a "goto" to the label or by falling through the program to the label. The notation SD indicates a statement or declaration. In PASCAL declarations are not allowed in blocks; however, the effect of permitting declarations within the block is the creation of a set of true propositions which can be used in subsequent proofs in that block.

At the opposite extreme several statement forms have difficult or unknown proof rules.

```
region E do S; ... endregion
retry
lock E
unlock E
failing SD; ... failhere SD; ... endfail
fail
fail finishing E
```

These statements all have a dynamic action which cannot be determined statically. The region and lock statements seem to have the effect of an assignment; however, stating the mutual exclusion property in axiom form seems difficult. The failing statement declares a dynamic structure within the statement and corresponding invoked functions and routines. These routines and functions may be separately compiled or compiled in separate scopes within this module. What is needed is a proposition which will be guaranteed to be true following a "fail" command execution.

The looping constructions are very similar to those used in PASCAL. They are slightly more complex because the COL contains "loop" and "break" commands to continue to next iteration or leave the loop. Rather than state the proof rules we will indicate the modifications to the looping proof rules of PASCAL. In PASCAL, each looping statement has a proof rule which involves a proposition called the loop invariant. This proposition and any information determined by the conditional test must be true before the loop body is executed and following the execution of the loop body. To include the "loop" statement in the loop involves proving that the truth of the proposition before the execution of "loop" implies the loop invariant. The "break" statement can be used as follows: Determine the weakest proposition which is implied by the propositions true before the execution of each "break" statement. Then the proposition true following the loop statement is the logical or of the loop invariant and this "break" proposition.

The "return" and "resultis" statements are the COL method of routine and function exit. The "return" statement is equivalent to a "goto" to the end of the routine body. This means that the proposition true before the "return" must imply the exit proposition from the routine. In other words, the exit proposition for a routine is the weakest proposition that is

implied by the propositions preceding each "return" statement. The "resultis" statement is handled similarly, except the expression E is returned as the value of the function.

The conditional statements "test" and "unless" are in the COL but not in PASCAL. The test statement is a generalization of the two-armed conditional. It is equivalent to:

```
if E then begin S; ... end
else if E1 then begin S1; ... end
...
else begin Sn; ... end
```

Thus the proof rule for "test" is a straightforward generalization of the proof rule for if-then-else. The "unless" statement is equivalent to the statement:

```
if not E then begin S; ... end
```

Thus its proof rule is a modification of the single branch if-then proof rule for PASCAL.

The "switchon" statement in the COL is the "case" statement in PASCAL. The proof rule for "upon" is derivable from the proof rule for "switchon" by its semantic definition. There is a major difficulty with this proof rule. To increase the space efficiency of COL programs one case of the "switchon" statement may merge with another case sharing common statements. The "stopswitch" statement terminates the execution of the "switchon"

statement. This dynamic behavior makes the proof rule difficult or impossible to write. Such bizarre behavior as entering a case in the middle of a loop is possible with the current semantics. The "switchon" and "upon" statements are related to the PASCAL case statement only when such bizarre behavior is not allowed. This is perhaps an aspect of the COL that might well be changed.

In summary the COL meets the security needs of programmers at a level equal to or exceeding the level of PASCAL. The following modifications to the COL should be studied to see if the security characteristics of the COL can be improved:

- add "uses" and "defines" lists to procedures
- study provability properties of "fail" and "lock"
- modify "switchon" and "upon" for clearer proof rules

[Hoare-69]
C.A.R. Hoare, "An Axiomatic Basis of Computer Programming", CACM Vol. 12 No. 10, pp. 576-580,583, October 1969

[Hoare-70]
C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach", in E. Engeler(ed.) "Lecture Notes in Mathematics 188", pp. 102-116, Springer-Verlag, 1970

[Clint]
M. Clint and C.A.R Hoare, "Program Proving: Jumps and Functions", Acta Informatica Vol. 1, pp.214-224, 1972

[Hoare-72]
C.A.R. Hoare, "Proof of Correctness of Data Representations", Acta Informatica Vol. 1, pp. 271-281, 1972

[Hoare-73]
C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica, Vol. 2, pp. 335-355, 1973

[Wirth]
Niklaus Wirth, "MODULA: A language for modular multiprogramming", Institut fur Informatik Report 18, Eidenossische Technische Hochschule, Zurich, Switzerland, March 1976

Chapter 16

Comparison of the COL with CS-4

16.1 Introduction

This chapter compares and contrasts the features of the COL with the language CS-4. CS-4 is an extensible language for "real-time, tactical environments and general purpose applications" described in the document CS-4 Language Reference Manual and Operating System Interface, Intermetrics, Inc., October, 1975.

This comparison is in five parts. Section 16.2 summarizes the declared purposes of the two languages, gives their current status, and presents the ground rules for comparison. Section 16.3 mentions the common features of the two languages and Sections 16.4 and 16.5 describe the features unique to CS-4 and COL, respectively. Finally, Section 16.6 summarizes the findings of the comparison.

16.2 Ground Rules for Comparison

In comparing two languages it is necessary to keep in mind the stated purposes of each. That is, it would be unfair to fault a language for not having a feature that it had never

intended to have and that, in fact, had been purposefully excluded from the language as being an unnecessary burden on the compiler and on the programmer trying to learn the language. Therefore, in any comparison of languages it is necessary to state explicitly the avowed goals of each.

The COL Manual can be paraphrased as follows concerning the orientation for the proposed language:

> We came to the conclusion that, at least from the language standpoint, the basic requirements for communications applications were in essence no different from the requirements of systems programming languages, although the relative emphasis for a COL differed to some extent. The generalized requirements of
> - very high performance;
> - very high reliability;
> - capability to interface with and manipulate specialized hardware;
> - high portability;
> - sophisticated data structures;
> - sophisticated control structures; and
> - specialized object computers.
>
> are requirements of both applications.
>
> Thus, while the requirements for efficient access to machine hardware, for example, or the need for object code optimization, might demand greater emphasis in a COL than in a systems programming language, our approach was to specify a COL which is also a good systems programming language, with emphasis on those language areas which are most relevant to the communications application.

In contrast to COL, CS-4 is a much larger language with a correspondingly greater scope of applicability. It seems apparent that the designers of both COL and CS-4 were aware of

the WOODENMAN Report which gave requirements for a proposed common Defense Department language. But, while COL has been intentionally restricted to satisfying only a subset of these requirements, namely those related to communications programming, CS-4 is much more ambitious in seemingly trying to satisfy them all. In fact, the design goals listed in the CS-4 Manual--reliability, efficiency, portability, understandability, and extensibility--only differ from those of the WOODENMAN Report in their emphasis on extensibilty.

The problem then becomes that of comparing a small language, whose emphasis is communications programming, with a large, general purpose language whose stated goal is to please as many people as possible through its extensibility features. This is like comparing apples and oranges (more accurately, apples and watermelons). Faced with this problem, an attempt will be made throughout to emphasize those features of one language which might prove valuable in helping the other language satisfy its own design goals.

Another qualification must be made on this comparison. The development of both languages is currently only in the design stage. Thus, there are probably many features of both languages which will eventually be discarded as impractical. No attempt

will be made in this document to validate the feasibility of a particular language feature, but it can be stated that the CS-4 language designers have recognized that:

> "The design of a language to meet this goal has proved to be a difficult task. There were many design problems encountered for which no good solutions existed....
>
> "There are two difficulties that have prevented the implementation and introduction of the full CS-4 language at this time:
> a) The compiler for full CS-4 would be so large that several years would be required for its implementation.
> b) Several problems in the language design still remain to be solved.
>
> "In view of the importance of establishing a new language in the time frame shorter than that required for full CS-4, a compatible subset of full CS-4 has been specified....
>
> "The CS-4 base language will be implemented first."

Nevertheless, the design of full CS-4, as specified in the CS-4 Manual, will be used in making the comparison.

With these qualifications and ground rules in mind, an attempt will now be made to compare the features of COL and CS-4.

16.3 Common Features of the Two Languages

Both COL and CS-4 have been designed with the criteria of the Department of Defense WOODENMAN Report in mind. Therefore, the two languages have many features in common. In particular,

both languages are block structured and emphasize the principles of strong typing, structured programming, and isolation of machine dependencies. Both languages speak of eliminating implicit coercions and logically significant defaults. Additionally, the languages have a similar format; both use the ASCII character set, have reserved keywords, and provide a two-faceted comment convention (for line terminating and multi-line comments).

CS-4 and COL include integer, floating point, boolean, character, and enumeration data types. Both can operate on arrays, structures, and strings. Arrays can be manipulated componentwise and the packing of data can be controlled. In addition, both feature a mechanism for abbreviating long type declarations, a way of grouping names together for external access, and the intention of providing intermodule type checking at load time.

Structured programming doctrine is professed in both CS-4 and COL. The usual range of control structures has been included: procedures, conditionals, case statements, and restricted gotos (jumping out of a block or into a control structure is prohibited). In addition, the languages have included a feature for breaking out of a loop and a way of jumping to the next loop iteration.

Some miscellaneous common features of the two languages are: evaluation of boolean expressions in short-circuit mode, both static and dynamic memory allocation, call-by-reference and call-by-value parameter passing, and a way of passing an arbitrarily long (variadic) sequence of arguments. In addition, both languages allow the choice of open or closed elaboration of procedures.

These two languages both support extensive compile-time features including conditional compilation, environment inquiries, constant folding, and compiler directives. Both languages have an extension mechanism and a way of accessing the features of the hardware with encapsulated assembly language instructions.

Although the features mentioned above have been provided by both languages, there are many differences in detail and also many instances where one language or the other has gone beyond the common ground in providing an extension to a common feature. These points are elaborated on in the succeeding sections. By and large, however, the languages have been developed in the same spirit and encompass many of the same features.

16.4 Unique CS-4 Capabilities

Keeping in mind that CS-4 and COL have different design goals, this section attempts to enumerate those features of CS-4 missing from COL. It is most often the case that such features make available a capability not appropriate in light of COL's limited domain of applicability.

As far as data types are concerned, CS-4 has fractions, varying strings, complex numbers, vectors, matrices, sets, and machine dependent MSTRUCTURES, above and beyond those types available in COL. In addition, the language's emphasis on extensibility allows any user-defined types. A data type is defined not only by specifying a set of possible values but also by listing the allowable operations on variables of that type. The language provides a uniform interface, across data types, to routines responsible for initializing, deactivating, assigning, comparing, selecting from, and performing I/O on a variable of a given type. This provides easy user control of such operations.

COL and CS-4 are similar in their control structure features. Some minor differences are as follows. CS-4 has UPDATE blocks (mutual exclusion regions), rather than interlock variables and the region statement. CS-4 has extensive, PL/I-like exception handling capabilities, including SIGNAL,

ABORT, and RESIGNAL statements. It also has a call-by-name parameter passing mechanism, allows both conditional and keyword parameters, and can restrict access to a parameter to READ, WRITE, or READ/WRITE.

CS-4 has extensive compile-time facilities that allow the following: certain programmers can be restricted from using gotos or SIGNALs, turning off run-time checking, mentioning absolute store locations, using assembly code or interfaces to other languages, or using numeric ranges above the portability limits. The RENAME statement allows the programmer to systematically rename external variables whose names conflict with local variables. In addition, there are many run-time checks that can be ENABLEd or DISABLEd.

The designers of CS-4 have done a comprehensive job of isolating machine dependencies. Any use of any variable in a machine dependent way requires that the variable have type MSTRUCTURE. Once this declaration has been made, the absolute location, physical layout, overlapping, size, and alignment can all be controlled. Machine instructions can be included by declaring an MPROCEDURE (which can have either an open or a closed elaboration). Variables can be placed in either the general or the floating point registers. External procedures can

be accessed and an argument passing mechanism can be specified (either that of FORTRAN, PL/I, or assembler). A full set of environment enquiry variables allows careful control of the range and precision of all numeric quantities.

Part II of the CS-4 Manual is a description of its operating system interface mechanism. Because none of its features are basic to COL, no details will be given other than to mention that it includes a file directory system with stream, record, random access, index sequential, and edited I/O, and a mechanism for interprocess I/O much like the pipes concept found in UNIX. Processes can be created, terminated, scheduled, resumed, delayed, or inhibited. UPDATE blocks can be used for altering shared variables.

Finally, CS-4 has several features used in its extension mechanism for altering the syntax and semantics of the language.

16.5 Unique COL Features

Features unique to COL come under the headings of efficiency, data structures, control structures, macros and compiler directives. Most deficiencies can be explained by COL's philosophy: if a feature cannot be expressed in an efficient, machine independent way, then leave it out of the language and provide it either with macros or with library routines.

COL has attempted to provide a great deal of flexibility and power in its data structures. In addition to those common features already mentioned, COL provides recursive data structures and interlock variables. The COL designers have invented the concept of implicit parameters for passing the actual run-time array bounds rather than simply allowing a "*" and expecting the calling routine to pass the actual bound as a separate parameter. In COL, as opposed to CS-4, strings may contain special characters such as carriage return and backspace. Also unique is COL's way of specifying numeric literal constants in any base.

Besides providing the common structured programming control structures, COL has the additional features of recursive procedures, Zahn's device (an event driven loop exiting mechanism), and a variety of keywords that in most instances eliminate the use of explicit boolean NOT operations in conditional statements and loop termination conditions (such as UNLESS for IF NOT). In addition, COL has separate mechanisms for routines (executed for their effect) and functions (executed for the values they return).

Of particular interest are COL's compile-time facilities. These include three types of macros (abbreviations for type

declarations, parameterized string substitution, and open routine elaboration), and a long list of compiler directives (minimize compilation time, execution time, or object program size; ignore, use, or verify assertions; check array subscripts and integer ranges; give directives to the loader; include a file in-line; force the compiler to treat an expression as being of a particular type; and pack an array of structures either in series or in parallel). In addition, COL provides a mechanism for querying the object computer concerning its characteristics, including word size, byte size, number of registers, and memory size.

Together, these unique features give COL programmers the ability to write efficient, succinct, and safe programs in the communications programming domain.

16.6 Summary

This section has attempted to compare and contrast the designs of two new programming languages, COL (Communications Oriented Language) and CS-4. It was pointed out that most of the differences between the two languages could be accounted for by their differences in design goals. CS-4 is a general purpose language that attempts to provide safe, reliable, and structured

solutions to current programming language design problems. Its intended audience is so large that many features must be included that will undoubtedly make the design of the compiler both tedious and difficult.

COL, on the other hand, is intended for a very strict range of applications, communications programming. As such, it has been kept small and particular emphasis has been placed on run-time efficiency and portability.

Naturally, these differences in design goals have led to rather large differences in the languages. This report has mentioned the many data types and the extension mechanism available in CS-4 and the compiler directives and code efficiency mechanisms available in COL. Of course, it remains to be seen how these disparate designs are eventually realized.